A No-Frills Introduction to Lua 5.1 VM Instructions

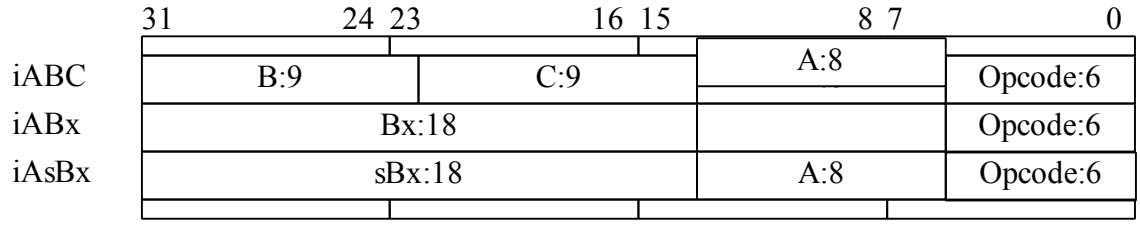
Lua 5.1 虚拟机指令简明手册

作者 **Kein-Hong Man, esq. <khman AT users.sf.net>**

版本 0.1, 20060313

Contents 目录

"A No-Frills Introduction to Lua 5.1 VM Instructions" is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License 2.0. You are free to copy, distribute and display the work, and make derivative works as long as you give the original author credit, you do not use this work for commercial purposes, and if you alter, transform, or build upon this work, you distribute the resulting work only under a license identical to



Lua 5 Instruction Formats

Lua5 指令格式

Instruction fields are encoded as simple unsigned integer values, except for sBx. Field sBx can represent negative numbers, but it doesn't use 2s complement. Instead, it has a bias equal to half the maximum integer that can be represented by its unsigned counterpart, Bx. For a field size of 18 bits, Bx can hold a maximum unsigned integer value of 262143, and so the bias is 131071 (calculated as `262143 >> 1`). A value of -1 will be encoded as (-1 + 131071) or 131070 or 1FFFE in hexadecimal.

除了 sBx，指令字段被编码为简单的无符号整型值。字段 sBx 可表示负数，但不是用 2 的补码，而是使用一个偏置（bias），它等于 sBx 的无符号等价物 Bx 可表示的最大整数的一半。对于 18 位的字段尺寸，Bx 能持有最大无符号整型值 262143，因而偏置是 131071（由 `262143 >> 1` 算出）。值-1 被编码为（-1 + 131071）或 131070 或十六进制的 1FFFE。

Fields A, B and C usually refers to register numbers (I'll use the term "register" because of its similarity to processor registers). Although field A is the target operand in arithmetic operations, this rule isn't always true for other instructions. A register is really an index into the current stack frame, register 0 being the bottom-of-stack position.

字段 A、B 和 C 通常引用寄存器编码（我将使用术语“寄存器”，因为它与处理器寄存器的相似性）。虽然在算术操作中字段 A 是目标操作数，但这个规则并非也适用于其他指令。寄存器通常是指向当前栈帧中的索引，0 号寄存器是栈底位置。

Unlike the Lua C API, negative indices (counting from the top of stack) are not supported. For some instructions, where the top of stack may be required, it is encoded as a special operand value, usually 0. Local variables are equivalent to certain registers in the current stack frame, while dedicated opcodes allow read/write of globals and upvalues. For some instructions, a value in fields B or C may be a register or an encoding of the number of a constant in the constant pool. This will be described further in the section on instruction notation.

与 C API 不同的是，负索引（从栈顶开始计数）是不支持的。某些指令需要指定栈顶，则索引被编码为特定的操作数（通常是 0）。局部变量等价于当前栈帧中的某个寄存器，但是也有允许读/写全局（变量）和 upvlaue 的操作码。对于某些指令来说，字段 B 或 C 的值可能为寄存器或常量池中的常量的已编码的编号。这方面在关于指令标记的章节会更深入地论述。

By default, Lua has a maximum stack frame size of 250. This is encoded as MAXSTACK in llimits.h. The maximum stack frame size in turn limits the maximum number of locals per function, which is set at 200, encoded as LUAI_MAXVARS in luaconf.h. Other limits found in the same file include the maximum number of upvalues per function (60), encoded as LUAI_MAXUPVALUES, call depths, the minimum C stack size, etc. Also, with an sBx field of 18 bits, jumps and control structures cannot exceed a jump distance of about 131071.

缺省时，Lua 具有 250 的最大栈帧尺寸，在 llimits.h 中编码为 MAXSTACK。它进而限制了每函数的局部变量的最大数目，200，在 luaconf.h 中编码为 LUAI_MAXVARS。该文件中的其他限制包括每函数的最大 upvalue 数（60），编码为 LUAI_MAXUPVALUES，调用深度，最小 C 栈尺寸，等等。并且，囿于 18 位的 sBx 字段，跳转和控制结构不能超出大约 131071 的跳转距离。

A summary of the Lua 5.1 virtual machine instruction set is as follows:

下面是 Lua5.1 虚拟机指令集的摘要：

Opcode 操作码	Name 命名	Description 说明
0	MOVE	Copy a value between registers 在寄存器间拷贝值
1	LOADK	Load a constant into a register 把一常量载入寄存器
2	LOADBOOL	Load a boolean into a register 把一布尔值载入寄存器
3	LOADNIL	Load **nil** values into a range of registers 把 **nil** 载入一系列寄存器
4	GETUPVAL	Read an upvalue into a register 把一 upvalue 读入寄存器
5	GETGLOBAL	Read a global variable into a register 把一全局变量读入寄存器
6	GETTABLE	Read a table element into a register 把一表元素读入寄存器
7	SETGLOBAL	Write a register value into a global variable 把一寄存器值写入全局变量
8	SETUPVAL	Write a register value into an upvalue 把一寄存器值写入 upvalue
9	SETTABLE	Write a register value into a table element 把一寄存器值写入表元素
10	NEWTABLE	Create a new table 创建表
11	SELF	Prepare an object method for calling 为调用对象方法做准备
12	ADD	Addition operator 加法操作
13	SUB	Subtraction operator 减法操作
14	MUL	Multiplication operator 乘法操作
15	DIV	Division operator 除法操作
16	MOD	Modulus (remainder) operator 取模（余数）操作
17	POW	Exponentiation operator 取幂操作
18	UNM	Unary minus operator 一元负操作
19	NOT	Logical NOT operator 逻辑非操作
20	LEN	Length operator 取长度操作
21	CONCAT	Concatenate a range of registers 连接一系列寄存器
22	JMP	Unconditional jump 无条件跳转
23	EQ	Equality test 相等测试
24	LT	Less than test 小于测试
25	LE	Less than or equal to test 小于或等于测试
26	TEST	Boolean test, with conditional jump 布尔测试，带条件跳转
27	TESTSET	Boolean test, with conditional jump and assignment 布尔测试，带条件跳转和赋值
28	CALL	Call a closure 调用闭包
29	TAILCALL	Perform a tail call 执行尾调用
30	RETURN	Return from function call 从函数调用返回
31	FORLOOP	Iterate a numeric **for** loop 迭代数字 **for** 循环
32	FORPREP	Initialization for a numeric **for** loop 初始化数字 **for** 循环
33	TFORLOOP	Iterate a generic **for** loop 迭代一般形式的 **for** 循环
34	SETLIST	Set a range of array elements for a table 设置表的一系列数组元素
35	CLOSE	Close a range of locals being used as upvalues 关闭被用作 upvalue 的一系列局部变量
36	CLOSURE	Create a closure of a function prototype 创建一函数原型的闭包
37	VARARG	Assign vararg function arguments to registers 把可变数量参数赋给寄存器

3 Really Simple Chunks 十分简单的程序块

Before heading into binary chunk and virtual machine instruction details, this section will demonstrate briefly how ChunkSpy can be used to explore Lua 5 code generation. All the examples in this document were produced using the Lua 5.1 version of ChunkSpy found in the ChunkSpy 0.9.8 distribution.

在深入二进制块和虚拟机指令细节以前，本节将简要地展示如何用 ChunkSpy 查看 Lua5 的代码生成。本文中的所有示例都是用 0.9.8 版分发包中对应 Lua5.1 版的 ChunkSpy 产生的。

First, start ChunkSpy in interactive mode (user input is set in bold):

首先，以交互模式启动 ChunkSpy（用户输入为黑体）：

```
$ lua ChunkSpy.lua --interact
ChunkSpy: A Lua 5.1 binary chunk disassembler
Version 0.9.8 (20060307)  Copyright (c) 2004-2006 Kein-Hong Man
The COPYRIGHT file describes the conditions under which this
software may be distributed (basically a Lua 5-style license.)

Type 'exit' or 'quit' to end the interactive session. 'help' displays
this message. ChunkSpy will attempt to turn anything else into a
binary chunk and process it into an assembly-style listing.
A '\' can be used as a line continuation symbol; this allows multiple
lines to be strung together.

>
```

We'll start with the shortest possible binary chunk that can be generated:

我们将从能生成的尽可能最短的二进制块开始：

```
>do end
; source chunk: (interactive mode)
; x86 standard (32-bit, little endian, doubles)

; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
[1] return    0   1
; end of function
```

ChunkSpy will treat your keyboard input as a small chunk of Lua source code. The library function `string.dump()` is first used to generate a binary chunk string, then ChunkSpy will disassemble that string and give you a brief assembly language-style output listing.

ChunkSpy 将把你的键盘输入视作短小的 Lua 源代码块。首先用库函数 `string.dump()` 生成二进制块字符串，然后 ChunkSpy 会反汇编该串并给出简要的汇编语言样式的输出清单。

Some features of the listing: Comment lines are prefixed by a semicolon. The header portion of the binary chunk is not displayed with the brief style. Data or header information that isn't

an instruction is shown as an assembler directive with a dot prefix. `luac`-style comments are generated for some instructions, and the instruction location is in square brackets.

清单的一些特征：注释行以分号为前缀。摘要样式不显示二进制块的头部。非指令的数据或头部信息显示为以点号为前缀的汇编指令。某些指令会生成 `luac` 样式的注释，并且指令位置放在方括号中。

A "`do end`" generates a single RETURN instruction and does nothing else. There are no parameters, locals, upvalues or globals. For the rest of the disassembly listings shown in this document, we will omit some common header comments and show only the function disassembly part. Instructions will be referenced by its marked position, e.g. line [1]. Here is another very short chunk:

"`do end`"仅生成一个 RETURN 指令而且什么也不做。没有参数、局部变量、upvalue 或全局变量。在本文中剩下的反汇编清单中，我们将省略某些共同的头部，只显示函数反汇编部分。指令用其标记的位置引用，例如行[1]。另一个非常简短的块：

```
>return
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
[1] return     0   1
[2] return     0   1
; end of function
```

A RETURN instruction is generated for every **return** in the source. The first RETURN (line [1]) is generated by the **return** keyword, while the second RETURN (line [2]) is always added by the code generator. This isn't a problem, because the second RETURN never gets executed anyway, and only 4 bytes is wasted. Perfect generation of RETURN instructions requires basic block analysis, and it is not done because there is no performance penalty for an extra RETURN during execution, only a negligible memory penalty.

源代码中的每个 **return** 都生成一个 RETURN 指令。第一个 RETURN（行[1]）是由关键字 **return** 生成的，代码生成器总是添加第二个 RETURN（行[2]）。这没有问题，因为第二个 RETURN 从不会被执行，而且只浪费了 4 字节。生成理想的 RETURN 指令需要基本的块分析，并未这么做是因为执行期间额外的 RETURN 没有性能损失，只有而不足道的内存损失。

Notice in these examples, the minimum stack size is 2, even when the stack isn't used. The next snippet assigns a constant value of 6 to the global variable **a**:

在这些示例中要注意，最小的栈尺寸是 2，即使并未用到栈。下一个片段把常量值 6 赋给全局变量 **a**：

```
>a=6
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.const  "a"  ; 0
.const  6  ; 1
[1] loadk      0   1       ; 6
```

```
[2] setglobal  0   0       ; a
[3] return     0   1
; end of function
```

All string and number constants are pooled on a per-function basis, and instructions refer to them using an index value which starts from 0. Global variable names need a constant string as well, because globals are maintained as a table. Line [1] loads the value 6 (with an index to the constant pool of 1) into register 0, then line [2] sets the global table with the constant "a" (constant index 0) as the key and register 0 (holding the number 6) as the value.

所有字符串和数值常量都在函数的基础上进行池化（管理），指令使用从 0 开始的索引引用它们。全局变量名也需要常量字符串，因为全局变量是作为表维护的。行[1]把值 6（常量池中的索引为 1）载入 0 号寄存器，然后行[2]用常量"a"（常量索引 0）作为键、0 号寄存器（持有数值 6）作为值设置全局表。

If we write the variable as a local, we get:

如果我们把该变量写成局部的，会得到：

```
>local a="hello"
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.const  "hello"  ; 0
[1] loadk      0   0       ; "hello"
[2] return     0   1
; end of function
```

Local variables reside in the stack, and they occupy a stack (or register) location for the duration of their existence. The scope of a local variable is specified by a starting program counter location and an ending program counter location; this is not shown in a brief disassembly listing.

局部变量驻留在栈中，它们在生存期内占用 1 个栈（或寄存器）位置。局部变量的作用域通过起始程序计数器（pc）位置和截止程序计数器位置指定；摘要反汇编清单不显示这些。

The local table in the function tells the user that register 0 is variable **a**. This information doesn't matter to the VM, because it needs to know register numbers only – register allocation was supposed to have been properly done by the code generator. So LOADK in line [1] loads constant 0 (the string "hello") into register 0, which is the local variable **a**. A stripped binary chunk will not have local variable names for debugging.

函数中的局部（变量）表告诉用户 0 号寄存器是变量 **a**。该信息对虚拟机来说并不重要，因为它只需要知道寄存器编号－假定代码生成器已经恰当地做好了寄存器分配。所以行[1]中的 LOADK 把 0 号常量（字符串"hello"）载入 0 号寄存器，即局部变量 **a**。剥离过调试信息的二进制块不会带有局部变量名。

Some examples in the following sections have been further annotated with additional comments in parentheses. Please note that ChunkSpy will not generate such comments, nor

will it indent functions that are at different nesting levels. Next we will take a look at the structure of Lua 5.1 binary chunks.

后面章节中的一些示例用圆括号中的注释做了进一步的诠释。请注意，ChunkSpy 不会生成这样的注释，也不会缩进不同嵌套层次的函数。接下来我们要看看 Lua5.1 二进制块的结构。

4 Lua Binary Chunks Lua 二进制程序块

Lua can dump functions as binary chunks, which can then be written to a file, loaded and run. Binary chunks behave exactly like the source code from which they were compiled.

Lua 能把函数转储（dump）为二进制块，它可被写入文件、加载并运行。二进制块行为表现得与编译出它们的源代码非常相似。

A binary chunk consist of two parts: a header block and a top-level function. The header portion contains 12 elements:

一个二进制块由两部分组成：头部块和顶层函数。头部包含 12 个元素：

Header block of a Lua 5 binary chunk Lua5 二进制块的头部块
Default values shown are for a 32-bit little-endian platform with IEEE 754 doubles as the number format. The header size is always 12 bytes.
显示的缺省值是用于以 IEEE 754 双精度浮点数为数字格式的 32 位小尾字节序平台。头部尺寸总是 12 字节。

4 bytes	Header signature: ESC, “Lua” or 0x1B4C7561 • Binary chunk is recognized by checking for this signature 头部签名：ESC，“Lua”或 0x1B4C7561 • 二进制块通过检查该签名来识别
1 byte	Version number, 0x51 (81 decimal) for Lua 5.1 • High hex digit is major version number • Low hex digit is minor version number 版本号，0x51（十进制的 81）用于 Lua5.1 • 高十六进制位是主版本号，低十六进制位是次版本号
1 byte	Format version, 0=official version 格式版本，0=官方版本
1 byte	Endianness flag (default 1) 字节序标志（缺省是 1） • 0=big endian, 1=little endian 0=大尾，1=小尾
1 byte	Size of int (in bytes) (default 4) int 的尺寸（单位是字节，缺省是 4）
1 byte	Size of size_t (in bytes) (default 4) size_t 的尺寸（单位是字节，缺省是 4）
1 byte	Size of Instruction (in bytes) (default 4) Instruction 的尺寸（单位是字节，缺省是 4）
1 byte	Size of lua_Number (in bytes) (default 8) lua_Number 的尺寸（单位是字节，缺省是 8）
1 byte	Integral flag (default 0) 整数标志（缺省是 0） • 0=floating-point, 1=integral number type • 0=浮点数，1=整数类型

On an x86 platform, the default header bytes will be (in hex):
在 x86 平台上，缺省头部字节是（十六进制）：

```
1B4C7561 51000104 04040800
```

A Lua 5.1 binary chunk header is always 12 bytes in size. Since the characteristics of a Lua virtual machine is hard-coded, the Lua undump code checks all 12 of the header bytes to determine whether the binary chunk is fit for consumption or not. All 12 header bytes of the binary chunk must exactly match the header bytes of the platform, otherwise Lua 5.1 will refuse to load the chunk. The header is also not affected by endianness; the same code can be used to load the main header of little-endian or big-endian binary chunks. The data type of `lua_Number` is determined by the size of `lua_Number` byte and the integral flag together.

Lua5.1 二进制块的头部尺寸总是 12 字节。由于 Lua 虚拟机的特征是硬编码的，Lua 的 undump 代码检查头部的全部 12 字节来决定二进制块是否适合运行。二进制块头部的全部 12 字节都必须严格匹配平台的头部字节，否则 Lua5.1 将拒绝加载该块。头部也不受字节序的影响；同样的代码可用来加载小尾序或大尾序二进制块的总头部。`lua_Number` 的数据类型由 `lua_Number` 的字节尺寸和整数标志一起确定。

In theory, a Lua binary chunk is portable; in real life, there is no need for the undump code to support such a feature. If you need undump to load all kinds of binary chunks, you are probably doing something wrong. If however you somehow need this feature, you can try ChunkSpy's rewrite option, which allows you to convert a binary chunk from one profile to another.

理论上，Lua 二进制块是可移植的；现实中并不需要 undump 代码支持该特性。如果你需要 undump 加载所有种类的二进制块，可能某些事情你做错了。如果你确实需要这个特性，可以尝试用 ChunkSpy 的重写选项，这个选项允许你把二进制块从一个 profile 转换为另一个。

Anyway, most of the time there is little need to seriously scrutinize the header, because since Lua source code is usually available, a chunk can be readily compiled into the native binary chunk format.

总之，多数时候没必要详查头部，原因是，由于 Lua 源代码一般是可用的，可轻易地把代码块编译成原生二进制格式。

The header block is followed immediately by the top-level function or chunk:

头部块后面紧跟着顶层函数或代码块：

Function block of a Lua 5 binary chunk Lua5 二进制块的函数块
Holds all the relevant data for a function. There is one top-level function.
持有函数的所有相关的数据。看一个顶层函数。

String	source name 源代码名
Integer	line defined 定义开始行
Integer	last line defined 定义结束行
1 byte	number of upvalues upvalue 数量
1 byte	number of parameters 参数数量
1 byte	is_vararg flag (see explanation further below)

	Is_vararg 标志（见下面的进一步解释） • 1=VARARG_HASARG • 2=VARARG_ISVARARG • 4=VARARG_NEEDSARG
1 byte	maximum stack size (number of registers used) 最大栈尺寸（使用的寄存器数量）
List	list of instructions (code) 指令（代码）列表
List	list of constants 常量列表
List	list of function prototypes 函数原型列表
List	source line positions (optional debug data) 源码位置（可选的调试数据）
List	list of locals (optional debug data) 局部变量列表（可选的调试数据）
List	list of upvalues (optional debug data) upvalue 列表（可选的调试数据）

A function block in a binary chunk defines the prototype of a function. To actually execute the function, Lua creates an instance (or *closure*) of the function first. A function in a binary chunk consist of a few header elements and a bunch of lists. Debug data can be stripped.

二进制块中的函数块定义了函数原型。要实际执行函数，Lua 首先创建一个实例（或*闭包*）。二进制块中的函数由一些头部元素和列表组成。调试数据可被剥离。

A **String** is defined in this way:

字符串以这种方式定义：

All strings are defined in the following format:
所有字符串以下面的格式定义：

Size_t	String data size 字符串数据尺寸
Bytes	String data, includes a NUL (ASCII 0) at the end 字符串数据，包括结尾的 NUL(ASCII 0)

The string data size takes into consideration a NUL character at the end, so an empty string (“”) has 1 as the size_t value. A size_t of 0 means zero string data bytes; the string does not exist. This is often used by the source name field of a function.
字符串数据尺寸考虑到了结尾的 NUL 字符，所以空字符串（””）的 size_t 值是 1。size_t 为 0 表示 0 个字符串数据字节；该字符串并不存在。函数的源代码名字段经常用它。

The **source name** is usually the name of the source file from which the binary chunk is compiled. It may also refer to a string. This source name is specified only in the top-level function; in other functions, this field consists only of a **Size_t** with the value 0.

源代码名通常是编译出二进制块的源文件的名字。它也可引用字符串。只在顶层函数中指定源代码名；在其他函数中，该字段只由 0 值的 **Size_t** 构成。

The **line defined** and **last line defined** are the line numbers where the function prototype starts and ends in the source file. For the main chunk, the values of both fields are 0. The next two fields, the **number of upvalues** and the **number of parameters**, are self-explanatory, as is the **maximum stack size** field. The **is_vararg** field is a bit more complicated, though. These are all byte-sized fields.

定义开始行和**定义结束行**是函数原型在源代码中开始和结束的行号。主代码块的这两个字段都是 0。之后的两个字段，**upvalue 数量**和**参数数量**，同**最大栈尺寸**字段一样是不言自明的。不过 **is_vararg** 字段稍微复杂些。这些是全部的字节尺寸的字段。

The **is_vararg** flag comprise 3 bitfields. By default, Lua 5.1 defines the constant LUA_COMPAT_VARARG, allowing the table **arg** to be used in functions that are defined with a variable number of parameters (*vararg* functions.) The table **arg** itself is not counted in the number of parameters. For old style code that uses **arg**, **is_vararg** is 7. If the code within the vararg function uses **...** instead of **arg**, then **is_vararg** is 3 (the VARARG_NEEDSARG field is 0.) If 5.0.2 compatibility is compiled out, then **is_vararg** is 2.

is_vararg 标志包含 3 个位字段。缺省时，Lua5.1 定义常量 LUA_COMPAT_VARARG，允许表 **arg** 被用于带可变数量的参数（*vararg* 函数）定义的函数。表 **arg** 自身计入参数数量内。对于使用 **arg** 的旧式代码，**is_vararg** 是 7。如果内含 vararg 函数的代码使用 **...** 代替 **arg**，那么 **is_vararg** 是 3（字段 VARARG_NEEDSARG 是 0）。如果编译出了 5.0.2 兼容性，则 **is_vararg** 是 2.

To summarize, the flag VARARG_ISVARARG (2) is always set for vararg functions. If LUA_COMPAT_VARARG is defined, VARARG_HASARG (1) is also set. If **...** is not used within the function, then VARARG_NEEDSARG (4) is set. A normal function always has an **is_vararg** flag value of 0, while the main chunk always has an **is_vararg** flag value of 2.

来个总结，vararg 函数总是设置标志 VARARG_ISVARARG（2）。如果定义了 LUA_COMPAT_VARARG，也会设置 VARARG_HASARG（1）。如果函数内没用**...**，则设置 VARARG_NEEDSARG（4）。平常的函数总是具有 0 值的 **is_vararg** 标志，可是主代码块总是具有值为 2 的 **is_vararg** 标志。

After the function header elements comes a number of lists that store the information that makes up the body of the function. Each list starts with an **Integer** as a list size count, followed by a number of list elements. Each list has its own element format. A list size of 0 has no list elements at all.

函数头部元素后面出现的是许多列表，它们存储构成函数体的信息。每个列表以作为列表尺寸的 **Integer** 开始，后面跟着很多列表元素。每个列表都有自己的元素格式。0 尺寸的列表没有列表元素。

In the following boxes, a data type in square brackets, e.g. **[Integer]** means that there are multiple numbers of the element, in this case an integer. The count is given by the list size. Names in parentheses are the ones given in the Lua sources; they are data structure fields.

在下面的方框中，在方括号中的数据类型，例如[**Integer**]，表示若干个元素，该例中

是整型。数量由列表尺寸给出。圆括号中的名字是 Lua 源代码中的；它们是数据结构中的字段。

The first list is the instruction list, or the actual code to the function. This is the list of instructions that will actually be executed:

第一个列表是指令表，或者说函数的实际编码。这是将被实际执行的指令列表。

Instruction list 指令表
Holds list of instructions that will be executed. 持有将被执行的指令列表。

Integer	size of code (sizecode) 编码尺寸
[Instruction]	virtual machine instructions 虚拟机指令

The format of the virtual machine instructions was given in the last chapter. A RETURN instruction is always generated by the code generator, so the size of the instruction list should be at least 1. Next is the list of constants:

前一节给出了虚拟机指令格式。编码生成器总会生成 RETURN 指令，所以指令表的尺寸至少是 1。接下来是常量表。

Constant list 常量表
Holds list of constants referenced in the function (it's a constant pool.)
持有函数中引用的常量列表（它是个常量池。）

Integer	size of constant list (sizek) 常量表尺寸
[	
1 byte	type of constant (value in parentheses): 常量类型（圆括号中的值） • 0=LUA_TNIL, 1=LUA_TBOOLEAN, • 3=LUA_TNUMBER, 4=LUA_TSTRING
Const	the constant itself: this field does not exist if the constant type is 0; it is 0 or 1 for type 1; it is a **Number** for type 3, or a **String** for type 4. 常量本身：如果常量类型是 0 则不存在该字段；对于类型 1 是 0 或 1；对于类型 3 是**数字**；对于类型 4 是**字符串**。
]	

Number is the Lua number data type, normally an IEEE 754 64-bit double. **Integer**, **Size_t** and **Number** are all endian-sensitive; Lua 5.1 will not load a chunk whose endianness is different from that of the platform. Their sizes and formats are of course specified in the binary chunk header. The data type of **Number** is determined by its size byte and the integral flag. Boolean values are encoded as either 0 or 1.

Number 是 Lua 数值数据类型，通常是 IEEE 754 64 位双精度浮点数。**Integer**、**Size_t** 和 **Number** 都是字节序敏感的；Lua5.1 不会加载字节序与平台不同的（二进制）块。它们的尺寸和格式当然也在二进制块的头部指定了。**数字**的数据类型由其尺寸和整型

标志确定。布尔值编码为 0 或 1。

The function prototype list comes after the constant list:

函数原型表跟在常量表后：

Function prototype list 函数原型表
Holds function prototypes defined within the function.
持有函数中定义的函数原型。

Integer	size of function prototypes (sizep) 函数原型（表）的大小
[Functions]	function prototype data, or function blocks 函数原型数据，或者说函数块

Function prototypes or function blocks have the exact same format as the top-level function or chunk. However, function prototypes that isn't the top-level function do not have the source name field defined. In this way, function prototypes at different lexical scoping levels are defined and nested. In a complex binary chunk, the nesting may be several levels deep. A closure will refer to a function by its number in the list.

函数原型或这说函数块的格式同顶层函数或（二进制）块完全一样。但是，非顶层函数的函数原型没定义源代码名字段。这样，函数原型被定义和嵌套在不同的词法作用域层次。这种嵌套在复杂的二进制块中可能达到若干层的深度。闭包通过其在（函数）表中的编号引用函数。

The lists following the list of prototypes are optional. They contain debug information and can be stripped to save space. First comes the source line position list:

原型列表后面的列表是可选的。它们含有调试信息，可以被剥离以节省空间。首先是源代码行位置表。

Source line position list 源代码行位置表
Holds the source line number for each corresponding instruction in a function. This information is used by error handlers or debuggers. In a stripped binary, the size of this list is zero. The execution of a function does not depend on this list.
持有函数中的每个指令相应的源代码行号。该信息用于错误处理器和调试器。在剥离处理过的二进制（块）中，该表尺寸为 0。函数的执行不依赖该表。

Integer	size of source line position list (sizelineinfo) 源代码行位置表的尺寸
[Integer]	list index corresponds to instruction position; the integer value is the line number of the Lua source where the instruction was generated 表索引对应指令位置；该整型值是生成指令的 Lua 源代码的行号。

Next up is the local list. Each local variable entry has 3 fields, a string and two integers:

接下来是局部（变量）表。每个局部变量项有 3 个字段，一个字符串和两个整数：

Local list 局部（变量）表
Holds list of local variable names and the program counter range in which the local variable is active.
持有局部变量名和程序计数器范围的列表，局部变量在该范围内是活动的。

Integer	size of local list (sizelocvars) 局部（变量）表的尺寸
[	
String	name of local variable (varname) 局部变量名
Integer	start of local variable scope (startpc) 局部变量作用域起点
Integer	end of local variable scope (endpc) 局部变量作用域终点
]	

The final list is the upvalue list:

最后的列表是 upvalue 表：

Upvalue list upvalue 表
Holds list of upvalue names.
持有 upvalue 名字的列表。

Integer	size of upvalue list (sizeupvalues) upvalue 表的尺寸
[String]	name of upvalue upvalue 的名字

All the lists are not shared or re-used: Locals, upvalues, constants and prototypes referenced in the code must be specified in the respective lists in the same function. In addition, locals, upvalues, constants and the function prototypes are indexed using numbers starting from 0. In disassembly listings, both the source line position list and the instruction list are indexed starting from 1. Note that the latter is by convention only; the indices does not matter to the virtual machine itself, since all jump-related instructions use only signed displacements. However, for debug information, the scope of local variables is encoded using absolute program counter positions, and these positions are based on a starting index of 1. This is also consistent with the output listing from `luac`.

所有列表都不可共享或重用：代码中引用的局部（变量）、upvalue、常量和原型必须在同一函数中的各自的列表中指定。另外，局部（变量）、upvalue、常量和函数原型用从 0 开始的数字索引。在反汇编清单中，源代码行位置表和指令表都从 1 开始索引。注意，下面（所说）的只是约定；索引不影响虚拟机本身，因为所有跳转相关的指令只用有符号位移。然而，对于调试信息，局部变量的作用域使用绝对程序计数器位置编码，而且这些位置基于起始索引 1.这与 `luac` 的输出清单是一致的。

How does it all fit in? You can easily generate a detailed binary chunk disassembly using ChunkSpy. Enter the following short bit of code and name the file `simple.lua`:

它们是如何装配在一起的？你可用 ChunkSpy 生成详细的二进制块反汇编。输入下面的简短代码并命名文件为 simple.lua：

```
local a = 8
function b(c) d = a + c end
```

Next, run ChunkSpy from the command line to generate the listing:

接着，从命令行运行 ChunkSpy 生成清单：

```
$ lua ChunkSpy.lua --source simple.lua > simple.lst
```

The following is a description of the generated listing (simple.lst), split into segments.

下面是生成的清单（simple.lst）的说明，已经分成了片段。

```
Pos   Hex Data          Description or Code
------------------------------------------------------------------------
0000                   ** source chunk: simple.lua
                       ** global header start **
0000  1B4C7561           header signature: "\27Lua"
0004  51                 version (major:minor hex digits)
0005  00                 format (0=official)
0006  01                 endianness (1=little endian)
0007  04                 size of int (bytes)
0008  04                 size of size_t (bytes)
0009  04                 size of Instruction (bytes)
000A  08                 size of number (bytes)
000B  00                 integral (1=integral)
                       * number type: double
                       * x86 standard (32-bit, little endian, doubles)
                       ** global header end **
```

This is an example of a binary chunk header. ChunkSpy calls this the global header to differentiate it from a function header. For binary chunks specific to a certain platform, it is easy to match the entire header at one go instead of testing each field. As described previously, the header is 12 bytes in size, and needs to be exactly compatible with the platform or else Lua 5.1 won't load the binary chunk.

这是二进制块头部的示例。ChunkSpy 把它称为全局头部以区别于函数头部。对于特定于某平台的二进制块，很容易一次性匹配整个头部而非测试每个字段。如前所述，头部尺寸是 12 字节，而且需要与平台严格相容，否则 Lua5.1 不会加载二进制块。

The global header is followed by the function header of the top-level function:

全局头部后面是顶层函数的函数头部：

```
000C                   ** function [0] definition (level 1)
                       ** start of function **
000C  0B000000           string size (11)
0010  73696D706C652E6C+  "simple.l"
0018  756100             "ua\0"
                       source name: simple.lua
001B  00000000           line defined (0)
001F  00000000           last line defined (0)
```

```
0023  00              nups (0)
0024  00              numparams (0)
0025  02              is_vararg (2)
0026  02              maxstacksize (2)
```

A function's header is always variable in size, due to the **source name** string. The source name is only present in the top-level function. A top-level chunk does not have a line number on which it is defined, so both the line defined fields are 0. There are no upvalues or parameters. A top-level chunk can always take a variable number of parameters; **is_vararg** is always 2 for the top-level chunk. The stack size is set at the minimum of 2 for this very simple chunk.

由于**源代码名**字符串的关系，函数头部的尺寸是可变的。源代码名只在顶层函数中存在。顶层块没有关于它在哪儿定义的行号，所以两个定义行字段都是 0.此处没有 upvalue 和参数。顶层块总是带有可变数量的参数；对顶层块来说 **is_vararg** 总是 2。这个非常简单的块的栈尺寸设为最小值 2。

Next we come to the various lists, starting with the code listing of the main chunk:

接下来看各种列表，从主块的编码列表开始：

```
                    * code:
0027  05000000        sizecode (5)
002B  01000000        [1] loadk     0   0       ; 8
002F  64000000        [2] closure   1   0       ; 1 upvalues
0033  00000000        [3] move      0   0
0037  47400000        [4] setglobal 1   1       ; b
003B  1E008000        [5] return    0   1
```

The first line of the source code compiles to a single instruction, line [1]. Local **a** is register 0 and the number 8 is constant 0. In line [2], an instance of function prototype 0 is created, and the closure is temporarily placed in register 1. The MOVE instruction in line [3] is actually used by the CLOSURE instruction to manage the upvalue **a**; it is not really executed. This will be explained in detail in Chapter 14. The closure is then placed into the global **b** in line [4]; "b" is constant 1 while the closure is in register 1. Line [5] returns control to the calling function. In this case, it exits the chunk.

源代码的第一行编译成单条指令，行[1]。局部（变量）**a** 是 0 号寄存器，数字 8 是 0 号常量。行[2]中创建了 0 号函数原型的实例，该闭包临时置于 1 号寄存器中。行[3]中的 MOVE 指令实际上是 CLOSURE 指令用来管理 upvalue **a** 的；它不会真的执行。这将在 14 节中详细解释。然后在行[4]中该闭包被置于全局（变量）**b** 中；"b"是 1 号常量，闭包在 1 号寄存器中。行[5]返回控制权给主调函数。此处是退出程序块。

The list of constants follow the instructions:

常量表在指令后面：

```
                    * constants:
003F  02000000        sizek (2)
0043  03              const type 3
0044  0000000000002040   const [0]: (8)
004C  04              const type 4
```

```
004D  02000000          string size (2)
0051  6200              "b\0"
                      const [1]: "b"
```

The top-level function requires two constants, the number 8 (which is used in the assignment on line 1) and the string “b” (which is used to refer to the global variable **b** on line 2.)

顶层函数需要两个常量，数字 8（用在行 1 的赋值中）和字符串“b”（用于行 2 引用全局变量 **b**）。

This is followed by the function prototype list of the main chunk. On line 2 of the source, a function prototype was declared within the main chunk. This function is instantiated and the closure is assigned to global **b**.

再往后是主程序块的函数原型表。在源代码的第 2 行上，主程序块声明了一个函数原型。该函数被实例化且其闭包被赋给全局（变量）**b**。

The function prototype list holds all the relevant information, a function block within a function block. ChunkSpy reports it as function prototype number 0, at level 2. Level 1 is the top-level function; there is only one level 1 function, but there may be more than one function prototype at other levels.

函数原型表持有函数块中的函数块的所有相关信息。ChunkSpy 把它报告为层次 2 上的 0 号函数原型。层 1 是顶层函数；只能有一个层 1 的函数，但是其他层次上可存在多于一个的函数原型。

```
                      * functions:
0053  01000000          sizep (1)

0057                   ** function [0] definition (level 2)
                      ** start of function **
0057  00000000          string size (0)
                      source name: (none)
005B  02000000          line defined (2)
005F  02000000          last line defined (2)
0063  01                nups (1)
0064  01                numparams (1)
0065  00                is_vararg (0)
0066  02                maxstacksize (2)
                      * code:
0067  04000000          sizecode (4)
006B  44000000          [1] getupval   1   0       ; a
006F  4C008000          [2] add       1   1   0
0073  47000000          [3] setglobal  1   0       ; d
0077  1E008000          [4] return     0   1
```

Above is the first section of function **b**’s prototype. It has no name string; it is defined on line 2 (both values point to line 2); there is one upvalue; there is one parameter, **c**; it is not a vararg function; and its maximum stack size is 2. Parameters are located from the bottom of the stack, so the single parameter **c** of the function is at register 0.

上面是函数 **b** 的原型的第一部分。它没有名字字符串；它在行 2 上定义（两个值都指向行 2）；有一个 upvalue；一个参数，**c**；不是 vararg 函数；最大栈尺寸是 2。参数位

于栈底开始的位置，所以该函数仅有的一个参数 **c** 在 0 号寄存器。

The prototype has 4 instructions. Most Lua virtual machine instructions are easy to decipher, but some of them have details that are not immediately evident. This example however should be quite easy to understand. In line [1], 0 is the upvalue **a** and 1 is the target register, which is a temporary register. Line [2] is the addition operation, with register 1 holding the temporary result while register 0 is the function parameter **c**. In line [3], the global **d** (so named by constant 0) is set, and in the next line, control is returned to the caller.

该原型有 4 条指令。多数 Lua 虚拟机指令是易于辨认的，但是其中一些具有不甚明显的细节。不过这个例子应该很好理解。在行[1]中，0 是 upvalue **a**，1 是目标寄存器，也是个临时寄存器。行[2]是加法操作，它用 1 号寄存器持有临时结果，而 0 号寄存器是函数参数 **c**。在行[3]中，全局变量 **d**（被 0 号常量如此命名）被设置，控制权在下一行被返回给调用者。

```
                    * constants:
007B  01000000        sizek (1)
007F  04             const type 4
0080  02000000        string size (2)
0084  6400            "d\0"
                    const [0]: "d"
                    * functions:
0086  00000000        sizep (0)
```

The constant list for the function has one entry, the string “d” is used to look up the global variable of that name. This is followed by the source line position list:

该函数的常量表有一项，字符串“b”用于查找同名全局变量。这之后是源代码行位置表。

```
                    * lines:
008A  04000000        sizelineinfo (4)
                    [pc] (line)
008E  02000000        [1] (2)
0092  02000000        [2] (2)
0096  02000000        [3] (2)
009A  02000000        [4] (2)
```

All four instructions that were generated came from line 2 of the source code.

生成的所有四条指令都来自与源代码的第 2 行。

The last two lists of the function prototype are the local list and the upvalue list:

函数原型的最后两个列表是局部（变量）表和 upvalue 表：

```
                    * locals:
009E  01000000        sizelocvars (1)
00A2  02000000        string size (2)
00A6  6300            "c\0"
                    local [0]: c
00A8  00000000          startpc (0)
00AC  03000000          endpc   (3)
                    * upvalues:
```

```
00B0  01000000          sizeupvalues (1)
00B4  02000000          string size (2)
00B8  6100             "a\0"
                      upvalue [0]: a
                      ** end of function **
```

There is one local variable, which is parameter **c**. For parameters, the `startpc` value is 0. Normal locals that are defined within a function have a `startpc` value of 1. There is also an upvalue, **a**, which refers to the local **a** in the parent (top) function.

这儿只有一个局部变量，即参数 **c**。对于参数来说，`startpc` 值是 0。函数汇总定义的常规局部变量 `startpc` 值为 1。这儿也有个 upvalue，**a**，它引用父（顶层）函数中的局部变量 **a**。

After the end of the function prototype data for function **b**, the chunk resumes with the debug information for the top-level chunk:

在函数 **b** 的函数原型数据末尾之后，程序块回到顶层程序块的调试信息：

```
                      * lines:
00BA  05000000          sizelineinfo (5)
                      [pc] (line)
00BE  01000000          [1] (1)
00C2  02000000          [2] (2)
00C6  02000000          [3] (2)
00CA  02000000          [4] (2)
00CE  02000000          [5] (2)
                      * locals:
00D2  01000000          sizelocvars (1)
00D6  02000000          string size (2)
00DA  6100             "a\0"
                      local [0]: a
00DC  01000000            startpc (1)
00E0  04000000            endpc   (4)
                      * upvalues:
00E4  00000000          sizeupvalues (0)
                      ** end of function **

00E8                   ** end of chunk **
```

From the source line list, we can see that there are 5 instructions in the top-level function. The first instruction came from line 1 of the source, while the other 4 instructions came from line 2 of the source.

从源代码行表我们能看出顶层函数有 5 条指令。第一条指令来自于源码行 1，而其他 4 条指令来自源码行 2。

The top-level function has one local variable, named “a”, active from program counter location 1 to location 4, and it refers to register 0. There are no upvalues, so the size of that table is 0. The binary chunk ends after the debug information of the main chunk is listed.

顶层函数有一个局部变量，名为“a”，从程序计数器位置 1 到位置 4 是活动的，而且它引用 0 号寄存器。此处没有 upvalue，所以该表尺寸为 0。二进制块在列出的主程序块的调试信息之后结束。

Now that we've seen a binary chunk in detail, we will proceed to look at each Lua 5.1 virtual machine instruction.

现在我们已经详细查看了二进制块，我们将继续探究 Lua5.1 虚拟机的每个指令。

5 Instruction Notation 指令记法

Before looking at some Lua virtual machine instructions, here is a little something about the notation used for describing instructions. Instruction descriptions are given as comments in the Lua source file `lopcodes.h`. The instruction descriptions are reproduced in the following chapters, with additional explanatory notes. Here are some basic symbols:

在查看 Lua 虚拟机指令以前，这儿有些关于描述指令的记法的东西。在 Lua 源码文件 `lopcodes.h` 中以注释方式给出了指令说明。后面的章节复制了这些指令说明，加了解释注记。这儿是一些基础符号：

R(A)	Register A (specified in instruction field A) 寄存器 A（由指令中的字段 A 指定）
R(B)	Register B (specified in instruction field B) 寄存器 B（由指令中的字段 B 指定）
R(C)	Register C (specified in instruction field C) 寄存器 C（由指令中的字段 C 指定）
PC	Program Counter 程序计数器
Kst(n)	Element **n** in the constant list 常量表中的元素 **n**
Upvalue[n]	Name of upvalue with index **n** 索引为 **n** 的 upvalue 的名字
Gbl[sym]	Global variable indexed by symbol **sym** 符号 **sym** 引用的全局变量
RK(B)	Register B or a constant index 寄存器 B 或常量索引
RK(C)	Register C or a constant index 寄存器 C 或常量索引
sBx	Signed displacement (in field sBx) for all kinds of jumps 适用于所有跳转到的有符号位移（字段 sBx 中）

The notation used to describe instructions is a little like pseudo-C. The operators used in the notation are largely C operators, while conditional statements use C-style evaluation. Booleans are evaluated C-style. Thus, the notation is a loose translation of the actual C code that implements an instruction.

用于描述指令的记法有点像伪 C。记法中用的操作符多半是 C 操作符，然而条件语句使用 C 式的求值。布尔也是 C 式求值。因此，该记法是实现指令的实际 C 代码的不严格的翻译。

The operation of some instructions cannot be clearly described by one or two lines of notation. Hence, this guide will supplement symbolic notation with detailed descriptions of the operation of each instruction. Having described an instruction, examples will be given to show the instruction working in a short snippet of Lua code. Using ChunkSpy's interactive mode, you can try out the examples yourself and get instant feedback in the form of disassembled code. If you want a disassembled listing plus the byte values of data and instructions, you can use ChunkSpy to generate a normal, verbose, disassembly listing.

一些指令的操作不能用以一两行标记清楚地描述。因此，本指南将补充一些象征性的标记，它们有对每个指令的操作的详细描述。介绍了指令之后会给出简短的 Lua 代码片段作为例子来展示指令的运作。利用 ChunkSpy 的交互模式，你能自己试验例子并

立即看到反汇编编码形式的反馈。如果你想得到反汇编清单外加数据和指令的字节值，可用 ChunkSpy 生成常规、详细的反汇编清单。

The program counter of the virtual machine (PC) always points to the next instruction. This behaviour is standard for most microprocessors. The rule is that once an instruction is read in to be executed, the program counter is immediately updated. So, to skip a single instruction following the current instruction, add 1 (the displacement) to the PC. A displacement of -1 will theoretically cause a JMP instruction to jump back onto itself, causing an infinite loop. Luckily, the code generator is not supposed to be able to make up stuff like that.

虚拟机的程序计数器（PC）总是指向下一条指令。这是多数微处理器上的标准行为。规则就是，一旦指令被读入并准备执行，程序计数器立刻更新。所以，要跳过当前指令后一条指令，给 PC 加 1（位移）。理论上，位移为-1 将使 JMP 指令跳回到自身（位置），引发无限循环。幸运的是，代码生成器应该不能编造那样的东西。

As previously explained, registers and local variables are roughly equivalent. Temporary results are always held in registers. Instruction fields B and C can point to a constant instead of a register for some instructions, this is when the field value has its MSB (most significant bit) set. For example, a field B value of 256 will point to the constant at index 0, if the field is 9 bits wide. For most instructions, field A is the target register. Disassembly listings preserve the A, B, C operand field order for consistency.

如前所述，寄存器和局部变量大致是等价的。临时结果总是在寄存器中存储。对某些指令来说，当指令字段 B 和 C 的值设置了 MSB（最高有效位）时，它们可指向常量而非寄存器。例如，如果字段 B 位宽度为 9，则值 256 将指向索引为 0 的常量。多数指令都把字段 A 用作目标寄存器。反汇编列表保持 A、B、C 操作数字段顺序的一致性。

6 Loading Constants 加载常量

Loads and moves are the starting point of pretty much all processor or virtual machine instruction sets, so we'll start with primitive loads and moves:

加载和移动是几乎全部处理器或虚拟机指令集的起点，所以我们从原始的装载和移动开始：

MOVE **A B** R(A) := R(B)

Copies the value of register R(B) into register R(A). If R(B) holds a table, function or userdata, then the *reference* to that object is copied. MOVE is often used for moving values into place for the next operation.
把寄存器 R(B)的值拷贝至寄存器 R(A)中。如果 R(B)持有一个表、函数或用户数据，则拷贝该对象的引用。MOVE 常用于为下一个操作移动值。

The opcode for MOVE has a second purpose – it is also used in creating closures, always appearing *after* the CLOSURE instruction; see CLOSURE for more information.
MOVE 的操作码还有第二个目的－它也被用于创建闭包，总是在 CLOSURE 指令之后出现；更多信息见 CLOSURE。

The most straightforward use of MOVE is for assigning a local to another local:
MOVE 最直接的用处是把一个局部变量赋给另一个局部变量：

```
>local a,b = 10; b = a
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.local  "b"  ; 1
.const  10  ; 0
[1] loadk     0   0       ; 10
[2] loadnil    1   1
[3] move      1   0
[4] return     0   1
; end of function
```

Line [3] assigns (*copies*) the value in local **a** (register 0) to local **b** (register 1).

行[3]把局部变量 **a**（0 号寄存器）中的值赋给（拷贝）局部变量 **b**（1 号寄存器）。

You won't see MOVE instructions used in arithmetic expressions because they are not needed by arithmetic operators. All arithmetic operators are in 2- or 3-operand style: the entire local stack frame is already visible to operands R(A), R(B) and R(C) so there is no need for any extra MOVE instructions.

你看不到 MOVE 指令被用在算术表达式中，因为算术操作符不需要它们。所有算术操作符都是 2 或 3 个操作数的形式：整个局部栈帧对 R(A)、R(B)和 R(C)都是可见的，所以不需要任何额外的 MOVE 指令。

Other places where you will see MOVE are:

- When moving parameters into place for a function call.
- When moving values into place for certain instructions where stack order is important, e.g. GETTABLE, SETTABLE and CONCAT.
- When copying return values into locals after a function call.
- After CLOSURE instructions (discussed in Chapter 14.)

能见到 MOVE 的其他地方是：

- 为函数调用移动参数时。
- 为某些栈顺序很重要的指令移动值，例如 GETTABLE、SETTABLE 和 CONCAT。
- 在函数调用后拷贝返回值到局部变量。
- 在 CLOSURE 指令后（在 14 节讨论）。

There are 3 fundamental instructions for loading constants into local variables. Other instructions, for reading and writing globals, upvalues and tables are discussed in the following chapters. The first constant loading instruction is LOADNIL:

有 3 条基本的指令用于加载常量到局部变量中。其他指令，如读写全局变量、upvalue 和表（的指令）在后面的章节讨论。第一个常量加载指令时 LOADNIL：

LOADNIL **A B** R(A) := ... := R(B) := nil

Sets a range of registers from R(A) to R(B) to **nil**. If a single register is to be assigned to, then R(A) = R(B). When two or more consecutive locals need to be assigned **nil** values, only a single LOADNIL is needed.
把从 R(A)到 R(B)范围的寄存器设为 **nil**。要赋给一个寄存器则 R(A) = R(B)。当两个或多个连续的局部变量需要赋给 **nil** 值时只需要一个 LOADNIL。

LOADNIL uses the operands A and B to mean a *range* of register locations. The example for MOVE in the last page shows LOADNIL used to set a single register to **nil**.

LOADNIL 用作数 A 和 B 表示一段范围的寄存器位置。上页中 MOVE 的例子显示 LOADNIL 被用于设置一个寄存器为 **nil**。

```
>local a,b,c,d,e = nil,nil,0
; function [0] definition (level 1)
; 0 upvalues, 0 params, 5 stacks
.function  0 0 2 5
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
.local  "d"  ; 3
.local  "e"  ; 4
.const  0  ; 0
[1] loadk     2   0       ; 0
[2] loadnil   3   4
[3] return    0   1
; end of function
```

Line [2] **nil**s locals **d** and **e**. A LOADNIL instruction is not needed for locals **a** and **b** because the instruction has been optimized away. Local **c** is explicitly initialized with the value 0. The

LOADNIL for locals **a** and **b** can be optimized away as the Lua virtual machine always sets all locals to **nil** prior to executing a function. The optimization rule is a simple one: If no other instructions have been generated, then a LOADNIL as the first instruction can be optimized away.

行[2]把局部变量 **d** 和 **e** 置 **nil**。局部变量 **a** 和 **b** 不需要 LOADNIL 指令，因为该指令已经被优化掉了。局部变量 **c** 显式地用 0 值初始化。局部变量 **a** 和 **b** 的 LOADNIL 能被优化掉是因为 Lua 虚拟机总是在执行函数以前先把全部局部变量设置为 **nil**。优化规则是很简单的一条：如果还没生成其他指令，则作为第一条指令的 LOADNIL 可被优化掉。

In the example, although the LOADNIL on line [2] is redundant, it is still generated as there is already an instruction that is not LOADNIL on line [1]. Ideally, one should put all locals that are initialized to **nil** at the top of the function, before anything else. In the above case, we can rearrange the locals to take advantage of the optimization rule:

在例子中，虽然行[2]上的 LOADNIL 是多余的，但因为行[1]上已经有了非 LOADNIL 的指令，所以仍然生成了它。理想状态下，应该把所有初始化为 **nil** 的局部变量放在函数顶部，在任何其他东西之前。在上面的情形中，我们可重新安排局部变量以利用优化规则：

```
>local a,b,d,e local c=0
; function [0] definition (level 1)
; 0 upvalues, 0 params, 5 stacks
.function  0 0 2 5
.local  "a"  ; 0
.local  "b"  ; 1
.local  "d"  ; 2
.local  "e"  ; 3
.local  "c"  ; 4
.const  0  ; 0
[1] loadk     4   0       ; 0
[2] return    0   1
; end of function
```

Now, we save one LOADNIL instruction. In other parts of a function, an explicit assignment of **nil** to a local variable will of course require a LOADNIL instruction.

现在我们节省了一条 LOADNIL 指令。在函数的其他部分，一个显式的给局部变量赋 **nil** 当然需要一条 LOADNIL 指令。

LOADK **A Bx** R(A) := Kst(Bx)
Loads constant number Bx into register R(A). Constants are usually numbers or strings. Each function has its own constant list, or pool. 把 Bx 号常量载入寄存器 R(A)。常量通常是数值或字符串。每个函数有自己的常量表，或者说池。

LOADK loads a constant from the constant list into a register or local. Constants are indexed starting from 0. Some instructions, such as arithmetic instructions, can use the constant list without needing a LOADK. Constants are pooled in the list, duplicates are eliminated. The

list can hold **nil**s, booleans, numbers or strings.

LOADK 把常量从常量表中载入寄存器或局部变量。常量从 0 开始索引。某些指令，例如算术指令，可无需 LOADK 直接使用常量表。常量实在列表中共用的，忽略复制。该列表可持有 **nil**、布尔、数值和字符串。

```
>local a,b,c,d = 3,"foo",3,"foo"
; function [0] definition (level 1)
; 0 upvalues, 0 params, 4 stacks
.function  0 0 2 4
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
.local  "d"  ; 3
.const  3  ; 0
.const  "foo"  ; 1
[1] loadk     0   0       ; 3
[2] loadk     1   1       ; "foo"
[3] loadk     2   0       ; 3
[4] loadk     3   1       ; "foo"
[5] return    0   1
; end of function
```

The constant 3 and the constant “foo” are both written twice in the source snippet, but in the constant list, each constant has a single location. The constant list contains the names of global variables as well, since GETGLOBAL and SETGLOBAL makes an implied LOADK operation in order to get the name string of a global variable first before looking it up in the global table.

在源码片段中常量 3 和“foo”都被写了两次，但在常量表中每个常量只有一个位置。常量表中也含有全局变量的名字，因为 GETGLOBAL 和 SETGLOBAL 隐式地做了 LOADK 操作以在全局表中查找它之前先得到全局变量的名字字符串。

The final constant-loading instruction is LOADBOOL, for setting a boolean value, and it has some additional functionality.

最后的常量加载指令是 LOADBOOL，用来设定布尔值，而且具有额外的功能。

LOADBOOL A B C R(A) := (Bool)B; if (C) PC++

Loads a boolean value (**true** or **false**) into register R(A). **true** is usually encoded as an integer 1, **false** is always 0. If C is non-zero, then the next instruction is skipped (this is used when you have an assignment statement where the expression uses relational operators, e.g. M = K>5.)
把布尔值（**true** 或 **false**）载入寄存器 R(A)中。**True** 通常编码为整数 1，**false** 总是 0。如果 C 非 0，则跳过下一条指令（用于其中的表达式使用了关系操作符的赋值语句，例如 M = K > 5）。

You can use any non-zero value for the boolean **true** in field B, but since you cannot use booleans as numbers in Lua, it’s best to stick to 1 for **true**.
你可在字段 B 中使用任何非 0 值作为布尔 **true**，但由于在 Lua 中不能把布尔用作数值，所以最好把 **true** 绑定到 1.

LOADBOOL is used for loading a boolean value into a register. It's also used where a boolean result is supposed to be generated, because relational test instructions, for example, do not generate boolean results – they perform conditional jumps instead. The operand C is used to optionally skip the next instruction (by incrementing PC by 1) in order to support such code. For simple assignments of boolean values, C is always 0.

LOADBOOL 用于把布尔值载入寄存器中。它也被用于期望生成布尔结果的地方，因为，例如关系测试指令并不生成布尔结果－而是执行条件跳转。操作数 C 用来选择性地跳过下一条指令（通过将 PC 增加 1）以支持这种编码。对于简单的赋布尔值，C 总是 0。

```
>local a,b = true,false
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.local  "b"  ; 1
[1] loadbool   0   1   0    ; true
[2] loadbool   1   0   0    ; false
[3] return     0   1
; end of function
```

This example is straightforward: Line [1] assigns **true** to local **a** (register 0) while line [2] assigns **false** to local **b** (register 1). In both cases, field C is 0, so PC is not incremented and the next instruction is not skipped.

这个例子很易懂：行[1]把 **true** 赋给局部变量 **a**（0 号寄存器），行[2]把 **false** 赋给局部变量 **b**（1 号寄存器）。两种情形中字段 C 都是 0，所以 PC 没有增加且下一条指令没有跳过。

```
>local a = 5 > 2
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.const  5  ; 0
.const  2  ; 1
[1] lt        1   257 256  ; 2 5, to [3] if false
[2] jmp       1          ; to [4]
[3] loadbool   0   0   1    ; false, to [5]
[4] loadbool   0   1   0    ; true
[5] return     0   1
; end of function
```

This is an example of an expression that gives a boolean result and is assigned to a variable. Notice that Lua does not optimize the expression into a **true** value; Lua 5.1 does not perform compile-time constant evaluation for relational operations, but it can perform simple constant evaluation for arithmetic operations.

在这个例子中，表达式给出布尔结果并赋给变量。注意，Lua 不会把表达式优化成一个 **true** 值；Lua5.1 并不为关系操作执行编译时的常量求值，但能为算数操作执行简单的常量求值。

Since the relational operator LT (which will be covered in greater detail later) does not give a boolean result but performs a conditional jump, LOADBOOL uses its C operand to perform an unconditional jump in line [3] – this saves one instruction and makes things a little tidier. The reason for all this is that the instruction set is simply optimized for **if**...**then** blocks. Essentially, `local a = 5 > 2` is executed in the following way:

由于关系操作符 LT（稍后将涉及更多细节）并不给出布尔结果而是执行条件跳转，LOADBOOL 用其 C 操作数在行[3]中执行无条件跳转－这省去一条指令并让事情稍微精简。所有这些的原因是，指令集只是为 **if**...**then** 程序块优化了。实质上，`local a = 5 > 2` 以下面的方式执行：

```
local a
if 2 < 5 then
  a = true
else
  a = false
end
```

In the disassembly listing, when LT tests 2 < 5, it evaluates to **true** and doesn't perform a conditional jump. Line [2] jumps over the false result path, and in line [4], the local **a** (register 0) is assigned the boolean **true** by the instruction LOADBOOL. If 2 and 5 were reversed, line [3] will be followed instead, setting a **false**, and then the true result path (line [4]) will be skipped, since LOADBOOL has its field C set to non-zero.

在反汇编清单中，当 LT 测试 2 < 5 时，它求得 **true** 并且不执行条件跳转。行[2]跳过 false 结果路径，在行[4]中，指令 LOADBOOL 把布尔 **true** 赋给局部变量 **a**（0 号寄存器）。如果反转 2 和 5，将改为执行行[3]设置 **false**，并且跳过 true 结果路径（行[4]），因为 LOADBOOL 的字段 C 被设为非 0。

So the true result path goes like this (additional comments in parentheses):

所以 true 结果路径如此进行（额外的注释在圆括号中）：

```
[1] lt        1   257 256  ; 2 5, to [3] if false (if 2 < 5)
[2] jmp       1          ; to [4]
[4] loadbool   0   1   0    ; true              (a = true)
[5] return     0   1
```

and the false result path (which never executes in this example) goes like this:

并且 false 结果路径（本例中决不会执行）如此进行：

```
[1] lt        1   257 256  ; 2 5, to [3] if false (if 2 < 5)
[3] loadbool   0   0   1    ; false, to [5]       (a = false)
[5] return     0   1
```

The true result path looks longer, but it isn't, due to the way the virtual machine is implemented. This will be discussed further in the section on relational and logic instructions.

true 结果路径看似长一些，根据虚拟机的实现方式，实际上并非如此。这些将在关于关系和逻辑指令的章节更深入地讨论。

7 Upvalues and Globals Upvalue 和全局变量

When the Lua virtual machine needs an upvalue or a global, there are dedicated instructions to load the value into a register. Similarly, when an upvalue or a global needs to be written to, dedicated instructions are used.

当 Lua 虚拟机需要 upvalue 或全局变量时，有转么的指令吧值载入寄存器中。类似地，当需要写入 upvalue 或全局变量时也用专门的指令。

GETGLOBAL A Bx R(A) := Gbl[Kst(Bx)]

Copies the value of the global variable whose name is given in constant number Bx into register R(A). The name constant must be a string.
把名为 Bx 号常量的全局变量的值拷贝到寄存器 R(A) 中。名字常量必须是字符串。

SETGLOBAL A Bx Gbl[Kst(Bx)] := R(A)

Copies the value from register R(A) into the global variable whose name is given in constant number Bx. The name constant must be a string.
把寄存器 R(A)的值拷贝到名为 Bx 号常量的全局变量中。名字常量必须是字符串。

The GETGLOBAL and SETGLOBAL instructions are very straightforward and easy to use. The instructions require that the global variable name be a constant, indexed by instruction field Bx. R(A) is either the source or target register. The names of the global variables used by a function will be part of the constant list of the function.

GETGLOBAL 和 SETGLOBAL 指令非常易懂且易用。它们要求全局变量名为指令字段 Bx 索引的常量。R(A)是源或目的寄存器。函数用到的全局变量的名字将是函数的常量表的一部分。

```
>a = 40; local b = a
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "b"  ; 0
.const  "a"  ; 0
.const  40  ; 1
[1] loadk     0   1       ; 40
[2] setglobal  0   0       ; a
[3] getglobal  0   0       ; a
[4] return    0   1
; end of function
```

From the example, you can see that “b” is the name of the local variable while “a” is the name of the global variable. Line [1] loads the number 40 into register 0 (functioning as a temporary register, since local **b** hasn’t been defined.) Line [2] assigns the value in register 0 to the global variable with name “a” (constant 0). By line [3], local **b** is defined and is assigned the value of global **a**.

从该例中，你能看到“b”是局部变量名而“a”是全局变量名。行[1]把数值 40 载入 0 号寄存器（功能是作为临时寄存器，因为局部变量 **b** 还未定义）。行[2]把 0 号寄存器中的值赋给名为“a” （0 号常量）的全局变量。行[3]定义了局部变量 **b** 并把全局变量 **a** 的值赋给它。

GETUPVAL A B R(A) := UpValue[B]

Copies the value in upvalue number B into register R(A). Each function may have its own upvalue list. This upvalue list is internal to the virtual machine; the list of upvalue name strings in a prototype is not mandatory.
把编号为 B 的 upvalue 拷贝到寄存器 R(A)中。每个函数可有自己的 upvalue 列表。该 upvalue 列表内置于虚拟机；原型中的 upvalue 名字字符串列表并不是强制的。

The opcode for GETUPVAL has a second purpose – it is also used in creating closures, always appearing *after* the CLOSURE instruction; see CLOSURE for more information.
GETUPVAL 的操作码有第二个目的一它也用于创建闭包，总是出现在 CLOSURE 指令后面；更多信息见 GLOSURE。

SETUPVAL A B UpValue[B] := R(A)

Copies the value from register R(A) into the upvalue number B in the upvalue list for that function.
从寄存器 R(A)拷贝其值到函数的 upvalue 列表中的编号为 B 的 upvalue。

GETUPVAL and SETUPVAL uses internally-managed upvalue lists. The list of upvalue name strings that are found in a function prototype is for debugging purposes; it is not used by the Lua virtual machine and can be stripped by `luac`.

GETUPVAL 和 SETUPVAL 使用内置管理的 upvalue 列表。函数原型中的 upvalue 名字字符串列表用作调试目的；它不是 Lua 虚拟机用的，而且能被 `luac` 剥离。

During execution, upvalues are set up by a CLOSURE, and maintained by the Lua virtual machine. In the following example, function **b** is declared inside the main chunk, and is shown in the disassembly as a function prototype within a function prototype. The indentation, which is not in the original output, helps to visually separate the two functions.

在执行期间，upvalue 是有 CLOSURE 设置并由 Lua 虚拟机维护的。在下面的例子中，函数 **b** 在主程序块内部声明，并且在反汇编中显示为函数原型内部的函数原型。缩进有助于直观地分隔两个函数，它并不是原始输出中的。

```
>local a; function b() a = 1 return a end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.const  "b"  ; 0

  ; function [0] definition (level 2)
  ; 1 upvalues, 0 params, 2 stacks
```

```
  .function  1 0 0 2
  .upvalue  "a"  ; 0
  .const  1  ; 0
  [1] loadk     0   0       ; 1
  [2] setupval   0   0       ; a
  [3] getupval   0   0       ; a
  [4] return     0   2
  [5] return     0   1
  ; end of function

[1] closure    1   0       ; 1 upvalues
[2] move       0   0
[3] setglobal  1   0       ; b
[4] return     0   1
; end of function
```

In the main chunk (function 0, level 1), local **a** starts as a **nil**. The CLOSURE in line [1] then instantiates function prototype 0 (function 0, level 2) with a single upvalue, **a**. Line [2] is part of the closure, it links local **a** in the current scope to upvalue **a** in the closure. Finally the closure is assigned to global **b**.

在主程序块中（1 级 0 号函数），局部变量 **a** 开始是 **nil**。行[1]中的 CLOSURE 实例化只带一个 upvalue，**a**，的 0 号函数原型（2 级 0 号函数）。行[2]是闭包的一部分，它把当前作用域中的局部变量 **a** 连接连接到闭包中的 upvalue **a**。最后闭包被赋给全局变量 **b**。

In function **b**, there is a single upvalue, **a**. In Pascal, a variable in an outer scope is found by traversing stack frames. However, instantiations of Lua functions are first-class values, and they may be assigned to a variable and referenced elsewhere. Moreover, a single prototype may have multiple instantiations. Managing upvalues thus becomes a little more tricky than traversing stack frames in Pascal. The Lua virtual machine solution is to provide a clean interface to access upvalues via GETUPVAL and SETUPVAL, while the management of upvalues is handled by the virtual machine itself.

在函数 **b** 中只有一个 upvalue，**a**。在 Pascal 中，外层作用域中的变量是通过遍历栈帧找到的。然而，Lua 函数实例是一类值，它们可被赋给变量在别处引用。此外，一个原型可有多个实例。因此 upvalue 的管理比 Pascal 中的遍历栈帧更有技巧性。Lua 虚拟机的解决办法是通过 GETUPVAL 和 SETUPVAL 提供访问 upvalue 的干净接口，不过 upvalue 的管理是虚拟机自己处理的。

Line [2] in function **b** sets upvalue **a** (upvalue number 0 in the upvalue table) to a number value of 1 (held in temporary register 0.) In line [3], the value in upvalue **a** is retrieved and placed into register 0, where the following RETURN instruction will use it as a return value. The RETURN in line [5] is unused.

函数 **b** 中的行[2]设置 upvalue **a**（upvalue 表中的 0 号 upvalue）为数值 1（存在临时的 0 号寄存器中）。在行[3]中，upvalue **a** 中的值被取回并放入 0 号寄存器，后面的 RETURN 指令将用它作为返回值。行[5]中的 RETURN 没用到。

8 Table Instructions 表指令

Accessing table elements is a little more complex than accessing upvalues and globals:

访问表元素比访问 upvalue 和全局变量稍微复杂些：

GETTABLE A B C R(A) := R(B)[RK(C)]

Copies the value from a table element into register R(A). The table is referenced by register R(B), while the index to the table is given by RK(C), which may be the value of register R(C) or a constant number.
拷贝表元素的值到寄存器 R(A)中。寄存器 R(B)引用表，而 RK(C)给出表引用，它可以是寄存器 R(C)的值或常量编号。

SETTABLE A B C R(A)[RK(B)] := RK(C)

Copies the value from register R(C) or a constant into a table element. The table is referenced by register R(A), while the index to the table is given by RK(B), which may be the value of register R(B) or a constant number.
从寄存器 R(C)或常量中拷贝值到表元素中。寄存器 R(A)引用表，而 RK(B)给出表索引，它可以是寄存器 R(B)的值或常量编号。

All 3 operand fields are used, and some of the operands can be constants. A constant is specified by setting the MSB of the operand to 1. If RK(C) need to refer to constant 1, the encoded value will be (256 | 1) or 257, where 256 is the value of bit 8 of the operand. Allowing constants to be used directly reduces considerably the need for temporary registers.

全部 3 个操作数都用到了，其中一些可谓常量。常量是通过把操作数的 MSB 设为 1 指定的。如果 RK(C)需要引用 1 号常量，编码后的值将是(256 | 1)或 257，其中 256 是操作数第 8 位的值。允许直接使用常量大大减少了对临时寄存器的需要。

```
>local p = {}; p[1] = "foo"; return p["bar"]
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "p"  ; 0
.const  1  ; 0
.const  "foo"  ; 1
.const  "bar"  ; 2
[1] newtable   0   0   0    ; array=0, hash=0
[2] settable   0   256 257  ; 1 "foo"
[3] gettable   1   0   258  ; "bar"
[4] return     1   2
[5] return     0   1
; end of function
```

In line [1], a new empty table is created and the reference placed in local **p** (register 0). Creating and populating new tables is a little involved so it will only be discussed later.

行[1]创建新表，其引用置于局部变量 **p**（0 号寄存器）中。创建和组装新表有点复杂，所以我们将在稍后讨论。

Table index 1 is set to “foo” in line [2] by the SETTABLE instruction. Both the index and the value for the table element are encoded constant numbers; 256 is constant 0 (the number 1) while 257 is constant 1 (the string “foo”.) The R(A) value of 0 points to the new table that was defined in line [1].

行[2]通过 SETTABLE 指令把表索引 1 设为“foo”。表元素的索引和值都被编码为常量编号；256 是 0 号常量（数值 1），257 是 1 号常量（字符串“foo”）。R(A)的值 0 指向行[1]中定义的新表。

In line [3], the value of the table element indexed by the string “bar” is copied into temporary register 1, which is then used by RETURN as a return value. 258 is constant 2 (the string “bar”) while 0 in field B is the reference to the table.

在行[3]中，字符串“bar”索引的表元素的值被拷贝到临时的 1 号寄存器中，它然后作为返回值用于 RETURN。258 是 2 号常量（字符串“bar”），字段 B 中的 0 是表的引用。

RK(B) and RK(C) type operands are also used in virtual machine instructions that implement binary arithmetic operators and relational operators.

RK(B)和 RK(C)类型的操作数也用于实现二元算数操作符和关系操作符的虚拟机指令中。

9 Arithmetic and String Instructions 算数和字符串指令

The Lua virtual machine's set of arithmetic instructions looks like 3-operand arithmetic instructions on an RISC processor. 3-operand instructions allow arithmetic expressions to be translated into machine instructions pretty efficiently.

Lua 虚拟机的算术指令集看起来就像 RISC 处理器上的 3 操作数的算术指令。3 操作数指令允许算术表达式非常高效地转化成机器指令。

ADD	**A B C**	R(A) := RK(B) + RK(C)
SUB	**A B C**	R(A) := RK(B) – RK(C)
MUL	**A B C**	R(A) := RK(B) * RK(C)
DIV	**A B C**	R(A) := RK(B) / RK(C)
MOD	**A B C**	R(A) := RK(B) % RK(C)
POW	**A B C**	R(A) := RK(B) ^ RK(C)

Binary operators (arithmetic operators with two inputs.) The result of the operation between RK(B) and RK(C) is placed into R(A). These instructions are in the classic 3-register style. RK(B) and RK(C) may be either registers or constants in the constant pool.
二元操作符（带两个输入的算术操作符）。在 RK(B)和 RK(C)间的操作结果放在 R(A)中。这些指令都是典型的 3 寄存器形式。RK(B)和 RK(C)可为寄存器或常量池中的常量。

ADD is addition. SUB is subtraction. MUL is multiplication. DIV is division. MOD is modulus (remainder). POW is exponentiation.
ADD 是加法。SUB 是减法。MUL 是乘法。DIV 是除法。MOD 是取模（余数）。POW 是求幂。

The source operands, RK(B) and RK(C), may be constants. If a constant is out of range of field B or field C, then the constant will be loaded into a temporary register in advance.

原操作数 RK(B)和 RK(C)可为常量。如果一个常量超出了字段 B 或字段 C 的范围，那么该常量将被预先载入临时寄存器中。

```
>local a,b = 2,4; a = a + 4 * b - a / 2 ^ b % 3
; function [0] definition (level 1)
; 0 upvalues, 0 params, 4 stacks
.function  0 0 2 4
.local  "a"  ; 0
.local  "b"  ; 1
.const  2  ; 0
.const  4  ; 1
.const  3  ; 2
[1] loadk     0   0       ; 2
[2] loadk     1   1       ; 4
[3] mul       2   257 1   ; 4    (loc2 = 4 * b)
[4] add       2   0   2          (loc2 = A + loc2)
[5] pow       3   256 1   ; 2    (loc3 = 2 ^ b)
[6] div       3   0   3          (loc3 = a / loc3)
[7] mod       3   3   258 ; 3    (loc3 = loc3 % 3)
[8] sub       0   2   3          (a = loc2 - loc3)
```

```
[9] return    0   1
; end of function
```

In the disassembly shown above, parts of the expression is shown as additional comments in parentheses. Each arithmetic operator translates into a single instruction. This also means that while the statement "`count = count + 1`" is verbose, it translates into a single instruction if **count** is a local. If **count** is a global, then two extra instructions are required to read and write to the global (GETGLOBAL and SETGLOBAL), since arithmetic operations can only be done on registers (locals) only.

在上面的反汇编中，表达式的部分作为额外的注释显示在圆括号中。每个算术操作符转为单条指令。这也意味着，尽管语句"`count = count + 1`"很冗长，如果 **count** 是局部变量则转为单条指令。如果 **count** 是全局变量，则需要两条额外的指令读写全局变量（GETGLOBAL 和 SETGLOBAL），因为算术操作符只能用于寄存器（局部变量）。

As of Lua 5.1, the parser and code generator can perform limited constant expression folding or evaluation. Constant folding only works for binary arithmetic operators and the unary minus operator (UNM, which will be covered next.) There is no equivalent optimization for relational, boolean or string operators.

从 Lua5.1 开始，解析器和编码生成器可执行有限的常量表达式折叠和求值。常量折叠只作用于二元操作符和一元负操作符（UNM，将在下一部分涉及）。没有相当的针对关系、布尔或字符串操作符的优化。

The optimization rule is simple: If both terms of a subexpression are numbers, the subexpression will be evaluated at compile time. However, there are exceptions. One, the code generator will not attempt to divide a number by 0 for DIV and MOD, and two, if the result is evaluated as a NaN (Not a Number) then the optimization will not be performed.

优化规则很简单：如果一个子表达式的两项都是数值，则该子表达式将在编译时求值。不过也有例外。第一个是，编码生成器不会处理 DIV 和 MOD 的数值被 0 除（的情况），第二个是，如果结果求值后为 NaN（非数值）则不会执行优化。

Also, constant folding is not done if one term is in the form of a string that need to be coerced. In addition, expression terms are not rearranged, so not all optimization opportunities can be recognized by the code generator. This is intentional; the Lua code generator is not meant to perform heavy duty optimizations, as Lua is a lightweight language. Here are a few examples to illustrate how it works (additional comments in parentheses):

并且，如果某一项是需要强制转换的字符串的形式则不会做常量折叠。另外，不会重新安排表达式项，所以编码生成器不能识别全部的优化机会。这是有意为之；Lua 编码生成器并没打算执行重量级的优化职能，因为 Lua 是个轻量级语言。看些说明它如何运转的例子（附加的注释在圆括号中 ）：

```
>local a = 4 + 7 + b; a = b + 4 * 7; a = b + 4 + 7
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
```

```
.const  "b"  ; 0
.const  11  ; 1
.const  28  ; 2
.const  4  ; 3
.const  7  ; 4
[1] getglobal  0   0       ; b
[2] add       0   257 0    ; 11    (a = 11 + b)
[3] getglobal  1   0       ; b
[4] add       0   1   258  ; 28    (a = b + 28)
[5] getglobal  1   0       ; b
[6] add       1   1   259  ; 4     (loc1 = b + 4)
[7] add       0   1   260  ; 7     (a = loc1 + 7)
[8] return     0   1
; end of function
```

For the first assignment statement, **4+7** is evaluated, thus 11 is added to **b** in line [2]. Next, in line [3] and [4], **b** and 28 are added together and assigned to **a** because multiplication has a higher precedence and **4*7** is evaluated first. Finally, on lines [5] to [7], there are two addition operations. Since addition is left-associative, code is generated for **b+4** first, and only after that, 7 is added. So in the third example, Lua performs no optimization. This can be fixed using parentheses to explicitly change the precedence of a subexpression:

对于第一条赋值语句，**4+7** 被求值，因此行[2]中 11 被加到 **b**。接着在行[3]和[4]中，**b** 和 28 被加到一起并赋给 **a**，因为乘法具有更高的优先级，**4*7** 首先求值。最后，在行[5]和[7]上有两个加法操作。由于加法是左结合的，所以首先为 **b+4** 生成编码，在那之后再加 7。所以在第三个例子中，Lua 没执行优化。这可通过使用圆括号显示地改变子表达式的优先级来修正。（译注－通过这个几个例子可看出，多次使用同一个全局变量不如先拷贝到局部变量；常量表中的项是经过预处理（优化）之后的，源代码中的常量不一定最后存在。）

```
>local a = b + (4 + 7)
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.const  "b"  ; 0
.const  11  ; 1
[1] getglobal  0   0       ; b
[2] add       0   0   257  ; 11
[3] return     0   1
; end of function
```

Now, the **4+7** subexpression can be evaluated at compile time. If the statement is written as:

现在，子表达式 **4+7** 可在编译时求值。如果语句写为：

```
local a = 7 + (4 + 7)
```

the code generator will generate a single LOADK instruction; Lua first evaluates **4+7**, then 7 is added, giving a total of 18. The arithmetic expression is completely evaluated in this case, thus no arithmetic instructions are generated.

编码生成器会生成单条 LOADK 指令；Lua 先对 **4+7** 求值，然后加上 **7**，给出总数

18。这样一来算术表达式被完全求值，因此没有生成算术指令。

In order to make full use of constant folding in Lua 5.1, the user just need to remember the usual order of evaluation of an expression's elements and apply parentheses where necessary. The following are two expressions which will not be evaluated at compile time:

为了充分利用 Lua5.1 的常量折叠，用户只需要记住常见的表达式元素的求值顺序并在需要的地方应用圆括号。下面是两个不会在编译时求值的表达式：

```
>local a = 1 / 0; local b = 1 + "1"
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.local  "b"  ; 1
.const  1  ; 0
.const  0  ; 1
.const  "1"  ; 2
[1] div       0   256 257  ; 1 0
[2] add       1   256 258  ; 1 "1"
[3] return    0   1
; end of function
```

The first is due to a divide-by-0, while the second is due to a string constant that needs to be coerced into a number. In both cases, constant folding is not performed, so the arithmetic instructions needed to perform the operations at run time are generated instead.

第一个似乎由于被 0 除，第二个是由于一个需要强制转为数值的字符串常量。这两种情况都不执行常量折叠，所以需要改为生成算术指令以在运行时执行这些操作。

Next are instructions for performing unary minus and logical NOT:

接下来是执行一元负和逻辑非的指令：

UNM **A B** R(A) := -R(B)

Unary minus (arithmetic operator with one input.) R(B) is negated and the value placed in R(A). R(A) and R(B) are always registers.
一元负（带一个输入的算术操作符）。对 R(B)取负并放入 R(A)中。R(A)和 R(B)总是寄存器。

NOT **A B** R(A) := not R(B)

Applies a boolean NOT to the value in R(B) and places the result in R(A). R(A) and R(B) are always registers.
对 R(B)中的值应用布尔 NOT 并把结果放入 R(A)中。R(A)和 R(B)总是寄存器。

Here is an example of both unary operations:

看下这两个一元操作的例子：

```
>local p,q = 10,false; q,p = -p,not q
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "p"  ; 0
.local  "q"  ; 1
.const  10  ; 0
[1] loadk     0   0       ; 10
[2] loadbool   1   0   0    ; false
[3] unm       2   0
[4] not       0   1
[5] move      1   2
[6] return    0   1
; end of function
```

Both UNM and NOT do not accept a constant as a source operand, making the LOADK on line [1] and the LOADBOOL on line [2] necessary. When an unary minus is applied to a constant number, the unary minus is optimized away. Similarly, when a **not** is applied to **true** or **false**, the logical operation is optimized away.

UNM 和 NOT 都不接受常量作为源操作数，需要产生行[1]的 LOADK 和行[2]的 LOADBOOL。当一元负被应用于常量数值时会被优化掉。类似地，当 **not** 被应用于 **true** 或 **false** 时该逻辑操作被优化掉。

In addition to this, constant folding is performed for unary minus, if the term is a number. So, the expression in the following is completely evaluated at compile time:

此外，如果项是数值，会为一元负执行常量折叠。所以，下面的表达式在编译时完全求值：

```
>local a = - (7 / 4)
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.const  -1.75  ; 0
[1] loadk     0   0       ; -1.75
[2] return    0   1
; end of function
```

Constant folding is performed on **7/4** first. Then, since the unary minus operator is applied to the constant 1.75, constant folding can be performed again, and the code generated becomes a simple LOADK (on line [1].)

首先在 **7/4** 上执行常量折叠。然后，由于一元负操作符被应用与常量 1.75，能再次执行常量折叠，生成的编码变成一条简单的 LOADK（在行[1]）。

LEN	**A B**	R(A) := length of R(B)
Returns the length of the object in R(B). For strings, the string length is returned, while for tables, the table size (as defined in Lua) is returned. For other objects, the metamethod is called. The result, which is a number, is placed in R(A).		

返回 R(B)中的对象的长度。字符串返回字符串长度，表返回表尺寸（在 Lua 中定义）。对其他对象调用元方法。结果是个数值，放在 R(A)中。

This instruction is new in Lua 5.1, implementing the **#** operator. If **#** operates on a constant, then the constant is loaded in advance using LOADK. The LEN instruction is currently not optimized away using compile time evaluation, even if it is operating on a constant string or table.

在 Lua5.1 中这条指令是新增的，实现了#操作符。如果#操作一个常量，则先用 LOADK 把常量载入。当前 LEN 指令不会用编译时求值优化掉，即使正在操作常量字符串或表。

```
>local a,b; a = #b; a= #"foo"
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "a"  ; 0
.local  "b"  ; 1
.const  "foo"  ; 0
[1] len       0   1
[2] loadk     2   0       ; "foo"
[3] len       0   2
[4] return    0   1
; end of function
```

In the above example, LEN operates on local **b** in line [1], leaving the result in local **a**. Since LEN cannot operate directly on constants, line [2] first loads the constant “foo” into a temporary local, and only then LEN is executed.

在上例中，在行[1]中 LEN 操作局部变量 **b**，把结果留在局部变量 **a** 中。由于 LEN 不能直接操作常量，行[2]先把常量“foo”载入临时局部变量，然后执行 LEN。

CONCAT **A B C** R(A) := R(B).. ... ..R(C)

Performs concatenation of two or more strings. In a Lua source, this is equivalent to one or more concatenation operators (‘**..**’) between two or more expressions. The source registers must be consecutive, and C must always be greater than B. The result is placed in R(A).
执行两个或更多字符串的连接。在 Lua 源代码中，这等价于两个或更多表达式之间的一个或更多的连接操作符（“..”）。源寄存器必须是连续的，C 必须总是比 Bda。结果放在 R(A)中。

Like LOADNIL, CONCAT accepts a range of registers. Doing more than one string concatenation at a time is faster and more efficient than doing them separately.

同 LOADNIL 一样，CONCAT 接受一组寄存器。一次做多个字符串连接比分开做更快更有效。

```
>local x,y = "foo","bar"; return x..y..x..y
; function [0] definition (level 1)
; 0 upvalues, 0 params, 6 stacks
```

```
.function  0 0 2 6
.local  "x"  ; 0
.local  "y"  ; 1
.const  "foo"  ; 0
.const  "bar"  ; 1
[1] loadk     0   0       ; "foo"
[2] loadk     1   1       ; "bar"
[3] move      2   0
[4] move      3   1
[5] move      4   0
[6] move      5   1
[7] concat    2   2   5
[8] return    2   2
[9] return    0   1
; end of function
```

In this example, strings are moved into place first (lines [3] to [6]) in the concatenation order before a single CONCAT instruction is executed in line [7]. The result is left in temporary local 2, which is then used as a return value by the RETURN instruction on line [8].

在该例中，在行[7]的单条 CONCAT 执行执行前，字符串首先被移动到顺序相连的地方。结果被留在临时的 2 号局部变量中，然后在行[8]被 RETURN 指令用作返回值。

```
>local a = "foo".."bar".."baz"
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "a"  ; 0
.const  "foo"  ; 0
.const  "bar"  ; 1
.const  "baz"  ; 2
[1] loadk     0   0       ; "foo"
[2] loadk     1   1       ; "bar"
[3] loadk     2   2       ; "baz"
[4] concat    0   0   2
[5] return    0   1
; end of function
```

In the second example, three strings are concatenated together. Note that there is no string constant folding. Lines [1] through [3] loads the three constants in the correct order for concatenation; the CONCAT on line [4] performs the concatenation itself and assigns the result to local **a**.

在第二个例子中，三个字符串被连接到一起。注意没有字符串折叠。行[1]到[3]为连接以正确的顺序载入三个常量；行[4]的 CONCAT 执行连接并把结果赋给局部变量 **a**。

10 Jumps and Calls 跳转和调用

Lua does not have any unconditional jump feature in the language itself, but in the virtual machine, the unconditional jump is used in control structures and logical expressions.

Lua 语言中没有任何无条件跳转的特性，但是在虚拟机中，无条件跳转被用于控制结构和逻辑表达式中。

JMP **sBx** PC += sBx

Performs an unconditional jump, with sBx as a signed displacement. sBx is added to the program counter (PC), which points to the next instruction to be executed. E.g., if sBx is 0, the VM will proceed to the next instruction.
以 sBx 为有符号位移执行无条件跳转。sBx 被加到程序计数器（PC）上，它指向将要执行的下一条指令。例如，如果 sBx 是 0，则 VM 将继续下一条指令。

JMP is used in loops, conditional statements, and in expressions when a boolean **true/false** need to be generated.
JMP 被用在循环、条件语句和需要生成布尔 **true/false** 的表达式中。

For example, since a relational test instruction makes conditional jumps rather than generate a boolean result, a JMP is used in the code sequence for loading either a **true** or a **false**:

例如，由于关系测试指令产生条件跳转而非生成布尔结果，JMP 被用在该编码序列中以载入 **true** 或 **false**。

```
>local m, n; return m >= n
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "m"  ; 0
.local  "n"  ; 1
[1] le        1   1   0    ; to [3] if false    (n <= m)
[2] jmp       1          ; to [4]
[3] loadbool   2   0   1    ; false, to [5]     (false path)
[4] loadbool   2   1   0    ; true             (true path)
[5] return     2   2
[6] return     0   1
; end of function
```

Line[1] performs the relational test. In line [2], the JMP skips over the false path (line [3]) to the true path (line [4]). The result is placed into temporary local 2, and returned to the caller by RETURN in line [5]. More examples where JMP is used will be covered in later chapters.

行[1]执行关系测试。在行[2]中，JMP 跳过 false 路径（行[3]）到 true 路径（行[4]）。结果被放在临时的 2 号局部变量中，并在行[5]中被 RETURN 返回给调用者。JMP 的更多例子将在后面的章节涉及。

Next we will look at the CALL instruction, for calling instantiated functions:

接着我们来看 CALL 指令，用来调用实例化的函数：

CALL **A B C** R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1))

Performs a function call, with register R(A) holding the reference to the function object to be called. Parameters to the function are placed in the registers following R(A). If B is 1, the function has no parameters. If B is 2 or more, there are (B-1) parameters.
执行函数调用，寄存器 R(A)持有要被调用的函数对象的引用。函数参数置于 R(A)之后的寄存器中。如果 B 是 1，函数没有返回值。如果 B 是 2 或更大则有（B-1）个参数。

If B is 0, the function parameters range from R(A+1) to the top of the stack. This form is used when the last expression in the parameter list is a function call, so the number of actual parameters is indeterminate.
如果 B 是 0，函数参数范围从 R(A+1)到栈顶。当参数表的最后一个表达式是函数调用时用这种形式，所以实际参数的数量是不确定的。

Results returned by the function call is placed in a range of registers starting from R(A). If C is 1, no return results are saved. If C is 2 or more, (C-1) return values are saved. If C is 0, then multiple return results are saved, depending on the called function.
函数调用的返回结果置于 R(A)开始的一组寄存器中。如果 C 是 1，不保存返回结果。如果 C 是 2 或更大则保存（C-1）个返回值。如果 C 是 0 则保存多个返回结果，依赖被调函数。

CALL always updates the top of stack value. CALL, RETURN, VARARG and SETLIST can use multiple values (up to the top of the stack.)
CALL 总是更新栈顶指针。CALL、RETURN、VARARG 和 SETLIST 可用多个值（直到栈顶）。

Generally speaking, for fields B and C, a zero means that multiple results or parameters (up to the top of stack) are expected. If the number of results or parameters are fixed, then the actual number is one less than the encoded field value. Here is the simplest possible call:

一般而言，对字段 B 和 C 来说，0 表示期望多个结果或参数（直到栈顶）。如果结果或参数的数量是确定的，则实际数量比编码后的字段值少一。这是可能最简单的调用：

```
>z()
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.const  "z"  ; 0
[1] getglobal  0   0       ; z
[2] call       0   1   1
[3] return     0   1
; end of function
```

In line [2], the call has zero parameters (field B is 1), zero results are retained (field C is 1), while register 0 temporarily holds the reference to the function object from global **z**. Next we see a function call with multiple parameters or arguments:

在行[2]中，调用具有 0 个参数（字段 B 是 1），保留 0 个结果（字段 C 是 1），同时

0 号寄存器持有来自全局变量 **z** 的函数对象。接下来我们看个带多个参数的函数调用：

```
>z(1,2,3)
; function [0] definition (level 1)
; 0 upvalues, 0 params, 4 stacks
.function  0 0 2 4
.const  "z"  ; 0
.const  1  ; 1
.const  2  ; 2
.const  3  ; 3
[1] getglobal  0   0       ; z
[2] loadk      1   1       ; 1
[3] loadk      2   2       ; 2
[4] loadk      3   3       ; 3
[5] call       0   4   1
[6] return     0   1
; end of function
```

Lines [1] to [4] loads the function reference and the arguments in order, then line [5] makes the call with an operand B value of 4, which means there are 3 parameters. Since the call statement is not assigned to anything, no return results need to be retained, hence field C is 1. Here is an example that uses multiple parameters and multiple return values:

行[1]到[4]按顺序载入函数引用和参数，然后行[5]以 4 为操作数 B 的值（表示有 3 个参数）开始调用。由于调用语句没赋给任何东西，没有返回值要保留，因此字段 C 是 1。这有个使用多个参数和多个返回值的例子：

```
>local p,q,r,s = z(y())
; function [0] definition (level 1)
; 0 upvalues, 0 params, 4 stacks
.function  0 0 2 4
.local  "p"  ; 0
.local  "q"  ; 1
.local  "r"  ; 2
.local  "s"  ; 3
.const  "z"  ; 0
.const  "y"  ; 1
[1] getglobal  0   0       ; z
[2] getglobal  1   1       ; y
[3] call       1   1   0
[4] call       0   0   5
[5] return     0   1
; end of function
```

First, the function references are retrieved (lines [1] and [2]), then function **y** is called first (temporary register 1). The CALL has a field C of 0, meaning multiple return values are accepted. These return values become the parameters to function **z**, and so in line [4], field B of the CALL instruction is 0, signifying multiple parameters. After the call to function **z**, 4 results are retained, so field C in line [4] is 5. Finally, here is an example with calls to standard library functions:

首先获取函数引用（行[1]和[2]），然后函数 **y** 首先被调用（临时的 1 号寄存器）。CALL 的字段 C 为 0，表示接受多个返回值。这些返回值成为函数 **z** 的参数，且在行[4]

中也是如此，CALL 指令的字段 B 是 0，表示多个参数。在函数 **z** 的调用之后保留 4 个结果，所以在行[4]中字段 C 是 5。最后一个例子是调用标准库函数：

```
>print(string.char(64))
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.const  "print"  ; 0
.const  "string"  ; 1
.const  "char"  ; 2
.const  64  ; 3
[1] getglobal  0   0       ; print
[2] getglobal  1   1       ; string
[3] gettable   1   1   258  ; "char"
[4] loadk      2   3       ; 64
[5] call       1   2   0
[6] call       0   0   1
[7] return     0   1
; end of function
```

When a function call is the last parameter to another function call, the former can pass multiple return values, while the latter can accept multiple parameters.

当函数调用是另一个函数调用的最后一个参数时，前一个能传递多个返回值，而后一个能接受多个参数。

Complementing CALL is RETURN:

与 CALL 互补的是 RETURN：

RETURN A B return R(A), ... ,R(A+B-2)

Returns to the calling function, with optional return values. If B is 1, there are no return values. If B is 2 or more, there are (B-1) return values, located in consecutive registers from R(A) onwards.
返回到主调函数，可选择带返回值。如果 B 是 1 则没有返回值。如果 B 是 2 或更大则有（B-1）个返回值，位于从 R(A)开始的连续寄存器中。

If B is 0, the set of values from R(A) to the top of the stack is returned. This form is used when the last expression in the return list is a function call, so the number of actual values returned is indeterminate.
如果 B 是 0，则返回从 R(A)开始到栈顶的值集。当返回列表中的最后一个表达式是函数调用时用该形式，所以实际返回值的数量是不确定的。

RETURN also closes any open upvalues, equivalent to a CLOSE instruction. See the CLOSE instruction for more information.
RETURN 也关闭任何打开的 upvalue，等价于 CLOSE 指令。更多信息见 CLOSE 指令。

Like CALL, a field B value of 0 signifies multiple return values (up to top of stack.)

同 CALL 一样，字段 B 的值为 0 表示多个返回值（直到栈顶）。

```
>local e,f,g; return f,g
; function [0] definition (level 1)
; 0 upvalues, 0 params, 5 stacks
.function  0 0 2 5
.local  "e"  ; 0
.local  "f"  ; 1
.local  "g"  ; 2
[1] move      3   1
[2] move      4   2
[3] return    3   3
[4] return    0   1
; end of function
```

In line [3], 2 return values are specified (field B value of 3.) The return values are placed in consecutive registers starting from register 3 by the MOVEs on line [1] and [2]. The RETURN in line [4] is redundant; it is always generated by the Lua code generator.

在行[3]中指定了 2 个返回值（字段 B 的值为 3）。行[1]和行[2]把返回值置于从 3 号寄存器开始的连续寄存器中。行[4]中的 RETURN 是多余的；Lua 编码生成器总是生成它。

TAILCALL A B C return R(A)(R(A+1), ... ,R(A+B-1))

Performs a tail call, which happens when a **return** statement has a *single* function call as the expression, e.g. `return foo(bar)`. A tail call is effectively a *goto*, and avoids nesting calls another level deeper. Only Lua functions can be tailcalled.
当 **return** 语句*只有*一个函数调用作为表达式时执行尾调用，例如，`return foo(bar)`。尾调用实际上是个 *goto*，并且避免了调用另一个更深的层次。只有 Lua 函数能被尾调用。

Like CALL, register R(A) holds the reference to the function object to be called. B encodes the number of parameters in the same manner as a CALL instruction.
同 CALL 一样，寄存器 R(A)持有要被调用的函数的引用。B 编码了参数数量，方式同 CALL 一样。

C isn't used by TAILCALL, since all return results are significant. In any case, Lua always generates a 0 for C, to denote multiple return results.
TAILCALL 不用（字段）C，因为所有返回值都是有意义的。无论如何，Lua 总是为 C 生成 0 以指示多返回值。

A TAILCALL is used only for one specific **return** style, described above. Multiple return results are always produced by a tail call. Here is an example:

TAILCALL 值用于上面描述的那种指定的 **return** 样式。尾调用总是产生多返回值。看个例子：

```
>return x("foo", "bar")
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
```

```
.const  "x"  ; 0
.const  "foo"  ; 1
.const  "bar"  ; 2
[1] getglobal  0   0       ; x
[2] loadk      1   1       ; "foo"
[3] loadk      2   2       ; "bar"
[4] tailcall   0   3   0
[5] return     0   0
[6] return     0   1
; end of function
```

Arguments for a tail call are handled in exactly the same way as arguments for a normal call, so in line [3], the tail call has a field B value of 3, signifying 2 parameters. Field C is 0, for multiple returns; this due to the constant LUA_MULTRET in `lua.h`. In practice, field C is not used by the virtual machine (except as an assert) since the syntax guarantees multiple return results.

尾调用处理参数的方式同常规调用完全一样，所以在行[4]中，尾调用的字段 B 值为 3，指示 2 个参数。对多返回来说字段 C 是 0；这取决于 `lua.h` 中的常量 LUA_MULTRET。实际上虚拟机没用到字段 C（除了一个断言），因为语法保证了多返回结果。

Line [5] is a RETURN instruction specifying multiple return results. This is required when the function called by TAILCALL is a C function. In the case of a C function, execution continues to line [5] upon return, thus the RETURN is necessary. Line [6] is redundant. When Lua functions are tailcalled, the virtual machine does not return to line [5] at all.

行[5]是个指定了多返回结果的 RETURN 指令。当 TAILCALL 调用的函数是 C 函数时需要这样。对 C 函数来说，执行绪持续到行[5]的 return 上面，因此 RETURN 是必需的。行[6]多余。当尾调用 Lua 函数时，虚拟机根本不返回到行[5]。

The other instructions covered in this section are SELF and VARARG. Both instructions are covered here because they are closely tied to function calls. We will start with VARARG:

本节要讲的其他指令是 SELF 和 VARARG。这两个指令之所以放在这儿是因为它们紧密地绑在函数调用上。我们从 VARARG 开始：

VARARG A B R(A), R(A+1), ..., R(A+B-1) = vararg

VARARG implements the vararg operator '**...**' in expressions. VARARG copies B-1 parameters into a number of registers starting from R(A), padding with **nil**s if there aren't enough values. If B is 0, VARARG copies as many values as it can based on the number of parameters passed. If a fixed number of values is required, B is a value greater than 1. If any number of values is required, B is 0.

VARARG 实现表达式中的 vararg 操作符“...”。VARARG 拷贝 B-1 个参数到 R(A)开始的许多寄存器中，如果值不够则用 **nil** 填充。如果 B 是 0，VARARG 基于传入的参数数量拷贝尽可能多的值。如果需要固定数量的值，则 B 是比 1 大的值。如果需要任意数量的值，则 B 是 0。（译注－上面的伪代码似乎不对，应该是 R(A), R(A+1), ..., R(A+B-2) = vararg。）

The use of VARARG will become clear with the help of a few examples:

通过几个例子的帮助会对 VARARG 的用法更清楚：

```
>local a,b,c = ...
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
[1] vararg    0   4
[2] return    0   1
; end of function
```

Note that the main or top-level chunk is a vararg function, as the **is_vararg** flag is set (the third number of the `.function` directive) in the example above. In this example, the left hand side of the assignment statement needs three values (or objects.) So in line [1], the operand B of the VARARG instruction is (3+1), or 4. VARARG will copy three values into **a**, **b** and **c**. If there are less than three values available, **nil**s will be used to fill up the empty places.

注意主程序或顶层程序块是 vararg 函数，因为上例中设置了 **is_vararg** 标志（`.function` 伪指令的第三个数值）。在本例中，赋值语句的左手边需要三个值（或对象）。所以在行[1]中 VARARG 指令的操作数 B 是（3+1），或 4.VARARG 将拷贝三个值到 **a**、**b** 和 **c** 中。如果可用的值不足三个将用 **nil** 填满空位置。

```
>local a = function(...) local a,b,c = ... end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0

  ; function [0] definition (level 2)
  ; 0 upvalues, 0 params, 4 stacks
  .function  0 0 3 4
  .local  "arg"  ; 0
  .local  "a"  ; 1
  .local  "b"  ; 2
  .local  "c"  ; 3
  [1] vararg    1   4
  [2] return    0   1
  ; end of function

[1] closure   0   0       ; 0 upvalues
[2] return    0   1
; end of function
```

Here is an alternate version where a function is instantiated and assigned to local **a**. The old-style **arg** is retained for compatibility purposes, but is unused in the above example.

这是个替代版本，其中实例化一个函数并赋给局部变量 **a**。出于兼容性目的保留旧式的 **arg**，但是上面的例子中并没使用。

```
>local a; a(...)
; function [0] definition (level 1)
```

```
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "a"  ; 0
[1] move      1   0
[2] vararg    2   0
[3] call      1   0   1
[4] return    0   1
; end of function
```

When a function is called with '**...**' as the argument, the function will accept a variable number of parameters or arguments. On line [2], a VARARG with a B field of 0 is used. The VARARG will copy all the parameters passed on to the main chunk to register 2 onwards, so that the CALL in the next line can utilize them as parameters of function **a**. The function call is set to accept a multiple number of parameters and returns zero results.

当函数被以"..."为参数被调用，它将接受数量可变的参数。在行[2]上，B 字段为 0 的 VARARG 被使用。VARARG 将把所有传入主程序块的参数拷贝至 2 号寄存器开始的（寄存器），所以下一行中的 CALL 指令可用它们作为函数 **a** 的参数。这个函数调用被设为接受可变数量的参数并返回 0 个结果。

```
>local a = {...}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
[1] newtable   0   0   0    ; array=0, hash=0
[2] vararg     1   0
[3] setlist    0   0   1    ; index 1 to top
[4] return     0   1
; end of function

>return ...
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
[1] vararg     0   0
[2] return     0   0
[3] return     0   1
; end of function
```

Above are two other cases where VARARG needs to copy all passed parameters over to a set of registers in order for the next operation to proceed. Both the above forms of table creation and **return** accepts a variable number of values or objects.

上面是两种其他情形，其中 VARARG 需要拷贝全部的传入参数至一集寄存器中，以便下一个操作的处理。上面两种形式的表创建和 **return** 接受可变数量的值或对象。

SELF **A B C** R(A+1) := R(B); R(A) := R(B)[RK(C)]

For object-oriented programming using tables. Retrieves a function reference from a table element and places it in register R(A), then a reference to the table itself is placed in the next register, R(A+1). This instruction saves some messy manipulation when setting up a method call.
面向对象程序设计使用表。从表元素中取回函数引用并放入寄存器 R(A)中，

然后把表本身的引用放在后续的寄存器 R(A+1)中。当准备方法调用时该指令省去了一些麻烦的操作。

R(B) is the register holding the reference to the table with the method. The method function itself is found using the table index RK(C), which may be the value of register R(C) or a constant number.
寄存器 R(B)持有该方法所在的表的引用。该方法自身是利用表索引 RK(C)找到的，后者可以是寄存器 R(C)的值或常量编号。

Finally, we have an instruction, SELF, which is used for object-oriented programming. A SELF instruction saves an extra instruction and speeds up the calling of methods in object-oriented programming. It is only generated for method calls that use the colon syntax. In the following example:

最后，我们有条指令，SELF，用于面向对象程序设计。SELF 指令省去了一条额外的指令并加速了面向对象程序设计中的方法调用。它只在使用了冒号语法的方法调用中才被生成。

```
>foo:bar("baz")
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.const  "foo"  ; 0
.const  "bar"  ; 1
.const  "baz"  ; 2
[1] getglobal  0   0       ; foo
[2] self       0   0   257  ; "bar"
[3] loadk      2   2       ; "baz"
[4] call       0   3   1
[5] return     0   1
; end of function
```

The method call is equivalent to: `foo.bar(foo, "baz")`, except that the global `foo` is only looked up *once*. This is significant if metamethods have been set. The SELF in line [2] is equivalent to a GETTABLE lookup (the table is in register 0 and the index is constant 1) and a MOVE (copying the table reference from register 0 to register 1.)

这个方法调用等价于 `foo.bar(foo, "baz")`，除了只查找一次全局变量 `foo`。如果设置了元方法这就很重要了。行[2]中的 SELF 等价于一次 GETTABLE 查找（表在 0 号寄存器中，索引是 1 号常量）和一次 MOVE（把表引用从 0 号寄存器拷贝至 1 号寄存器）。（译注－SELF 的结果是，方法覆盖了原先表所在的寄存器，因此后面的 CALL 的结果才能放在正确的位置，跟一般函数调用行为一致；另外，上面说的 GETTABLE 和 MOVE 的顺序应该调整。）

Without SELF, a GETTABLE will write its lookup result to register 0 (which the code generator will normally do) and the table reference will be overwritten before a MOVE can be done. Using SELF saves roughly one instruction and one temporary register slot.

如果没有 SELF，GETTABLE 将把查找结果写到 0 号寄存器（编码生成器的常规做法），这样在 MOVE 执行前表引用将被覆盖。使用 SELF 节省了大概一条指令和一个

临时寄存器位置。

After setting up the method call using SELF, the call is made with the usual CALL instruction in line [4], with two parameters. The equivalent code for a method lookup is compiled in the following manner:

使用 SELF 准备好方法调用后，在行[4]中常规的 CALL 指令产生带两个参数的调用。下面的方式中编译了方法查找的等价代码。

```
>foo.bar(foo, "baz")
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.const  "foo"  ; 0
.const  "bar"  ; 1
.const  "baz"  ; 2
[1] getglobal  0   0       ; foo
[2] gettable   0   0   257  ; "bar"
[3] getglobal  1   0       ; foo
[4] loadk      2   2      ; "baz"
[5] call       0   3   1
[6] return     0   1
; end of function
```

The alternative form of a method call is one instruction longer, and the user must take note of any metamethods that may affect the call. The SELF in the previous example replaces the GETTABLE on line [2] and the GETGLOBAL on line [3]. If **foo** is a local variable, then the equivalent code is a GETTABLE and a MOVE.

这种方法调用的方式多了一条指令，并且用户必须注意任何可能影响调用的元方法。前例中的 SELF 替换了行[2]的 GETTABLE 和行[3]的 GETGLOBAL。如果 **foo** 是局部变量，则等价的编码是 GETTABLE 和 MOVE。

Next we will look at more complicated instructions.

接下来我们来看看更多结构复杂的指令。

11 Relational and Logic Instructions 关系和逻辑指令

Relational and logic instructions are used in conjunction with other instructions to implement control structures or expressions. Instead of generating boolean results, these instructions conditionally perform a jump over the next instruction; the emphasis is on implementing control blocks. Instructions are arranged so that there are two paths to follow based on the relational test.

关系和逻辑指令与其他指令联合使用来实现控制结构或表达式。这些指令有条件地执行跳转来越过下一条指令，而不是生成布尔结果；其重点是在实现控制块上。

EQ	**A B C**	if ((RK(B) == RK(C)) ~= A) then PC++
LT	**A B C**	if ((RK(B) < RK(C)) ~= A) then PC++
LE	**A B C**	if ((RK(B) <= RK(C)) ~= A) then PC++

Compares RK(B) and RK(C), which may be registers or constants. If the boolean result is not A, then skip the next instruction. Conversely, if the boolean result equals A, continue with the next instruction.
比较 RK(B)和 RK(C)，它们可为寄存器或常量。如果布尔结果非 A 则跳过下一条指令。繁殖，如果布尔结果等于 A 则继续下一条指令。

EQ is for equality. LT is for “less than” comparison. LE is for “less than or equal to” comparison. The boolean A field allows the full set of relational comparison operations to be synthesized from these three instructions. The Lua code generator produces either 0 or 1 for the boolean A.
EQ 用于相等。LT 用于“小于”比较。LE 用于“小于或等于”比较。这三条指令中的布尔字段 A 允许与整个关系比较操作集合合成。Lua 编码生成器给布尔 A 产生 0 或 1。

For the fall-through case, a JMP is always expected, in order to optimize execution in the virtual machine. In effect, EQ, LT and LE must always be paired with a following JMP instruction.
为了优化虚拟机中的执行绪，对于失败的情况总是需要 JMP。实际上，EQ、LT 和 LE 必须总是与后跟的 JMP 指令配对。

By comparing the result of the relational operation with A, the sense of the comparison can be reversed. Obviously the alternative is to reverse the paths taken by the instruction, but that will probably complicate code generation some more. The conditional jump is performed if the comparison result is not A, whereas execution continues normally if the comparison result matches A. Due to the way code is generated and the way the virtual machine works, a JMP instruction is always expected to follow an EQ, LT or LE. The following JMP is optimized by executing it in conjunction with EQ, LT or LE.

通过关系操作结果与 A 的对照，比较的意义可能反转。很明显这种选择是为了反转指令的执行路径，但可能使编码生成稍微复杂。如果比较结果不是 A 则执行条件跳转，而如果比较结果与 A 匹配则执行绪继续（保持）正常。依据编码生成的方式和虚拟机运转的方式，总是需要在 EQ、LT 或 LE 后面跟着一条 JMP 指令。在跟 EQ、LT 或 LE 联合执行时，后面的 JMP 会被优化。

```
>local x,y; return x ~= y
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.local  "x"  ; 0
.local  "y"  ; 1
[1] loadnil    0   1
[2] eq        0   0   1    ; to [4] if true    (x ~= y)
[3] jmp       1           ; to [5]
[4] loadbool   2   0   1    ; false, to [6]     (false result path)
[5] loadbool   2   1   0    ; true             (true result path)
[6] return     2   2
[7] return     0   1
; end of function
```

In the above example, the equality test is performed in line [2]. However, since the comparison need to be returned as a result, LOADBOOL instructions are used to set a register with the correct boolean value. This is the usual code pattern generated if the expression requires a boolean value to be generated and stored in a register as an intermediate value or a final result.

上例中，行[2]中进行相等测试。不过，比较需要作为结果返回，所以用 LOADBOOL 指令把正确的布尔值设置到寄存器。如果表达式需要生成布尔值并存入寄存器作为中间值或最终值，这就是通常生成的编码模式。

It is easier to visualize the disassembled code as:

很容易把反汇编代码想象为：

```
if x ~= y then
  return true
else
  return false
end
```

The true result path (when the comparison result matches A) goes like this:

true 结果路径（当比较结果与 A 匹配时）像这样进行：

```
[1] loadnil    0   1
[2] eq        0   0   1    ; to [4] if true    (x ~= y)
[3] jmp       1           ; to [5]
[5] loadbool   2   1   0    ; true             (true path)
[6] return     2   2
```

while the false result path (when the comparison result does not match A) goes like this:

而 false 结果路径（当比较结果与 A 不匹配时）先这样进行：

```
[1] loadnil    0   1
[2] eq        0   0   1    ; to [4] if true    (x ~= y)
[4] loadbool   2   0   1    ; false, to [6]     (false path)
[6] return     2   2
```

ChunkSpy comments the EQ in line [2] by letting the user know when the conditional jump

is taken. The jump is taken when "the value in register 0 equals to the value in register 1" (the comparison) is not **false** (the value of operand A). If the comparison is **x == y**, everything will be the same except that the A operand in the EQ instruction will be 1, thus reversing the sense of the comparison. Anyway, these are just the Lua code generator's conventions; there are other ways to code **x ~= y** in terms of Lua virtual machine instructions.

ChunkSpy 在行[2]中给 EQ 作注释，让用户知道何时进行条件跳转。当"0 号寄存器的值等于 1 号寄存器的值"（比较）不为 **false**（操作数 A 的值）时进行跳转。如果比较是 **x == y**，则除了 EQ 指令的 A 操作数为 1 以外其他的都一样，这样反转了比较的意义。总之，这些只是 Lua 编码生成器的约定；就 Lua 虚拟机指令而言，还有其他的编码 **x ~= y** 的方式。

For conditional statements, there is no need to set boolean results. Lua is optimized for coding the more common conditional statements rather than conditional expressions.

没必要为条件语句设置布尔结果。Lua 为编码更一般的条件语句而非条件表达式进行了优化。

```
>local x,y; if x ~= y then return "foo" else return "bar" end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "x"  ; 0
.local  "y"  ; 1
.const  "foo"  ; 0
.const  "bar"  ; 1
[1] eq       1   0   1    ; to [3] if false    (x ~= y)
[2] jmp      3           ; to [6]
[3] loadk    2   0       ; "foo"             (true block)
[4] return   2   2
[5] jmp      2           ; to [8]
[6] loadk    2   1       ; "bar"             (false block)
[7] return   2   2
[8] return   0   1
; end of function
```

In the above conditional statement, the same inequality operator is used in the source, but the sense of the EQ instruction in line [1] is now reversed. Since the EQ conditional jump can only skip the next instruction, additional JMP instructions are needed to allow large blocks of code to be placed in both true and false paths. In contrast, in the previous example, only a single instruction is needed to set a boolean value. For **if** statements, the true block comes first followed by the false block in code generated by the code generator. To reverse the positions of the true and false paths, the value of operand A is changed.

在上面的条件语句中，源代码中用了同样的不等操作符，但是现在行[1]中的 EQ 指令的意义相反的。由于 EQ 条件跳转只能跳过下一条指令，需要额外的 JMP 指令以允许在 true 和 false 路径中放置大块的编码。相比之下，前例只需要单条指令来设置一个布尔值。对于 **if** 语句，在编码生成器生成的编码中 true 块先出现，后面跟着 false 块。改变操作数 A 的值就能反转 true 和 false 路径的位置。

The true path (when **x ~= y** is true) goes from [1] to [3]–[5] and on to [8]. Since there is a

RETURN in line [4], the JMP in line [5] and the RETURN in [8] are never executed at all; they are redundant but does not adversely affect performance in any way. The false path is from [1] to [2] to [6]–[8] onwards. So in a disassembly listing, you should see the true and false code blocks in the same order as in the Lua source.

true 路径（当 **x ~= y** 为 true 时）从[1]到[3]–[5]再到[8]。由于在行[4]中有 RETURN，行[5]中的 JMP 和[8]中的 RETURN 根本不会执行；它们是多余的但不会以任何方式对性能有负面影响。False 路径从[1]到[2]到[6]–[8]。所以在反汇编清单中，你应该看到 true 和 false 编码块是和 Lua 源码一样的顺序。

The following is another example, this time with an **elseif**:

下面是另一个例子，这次带有一个 **elseif**：

```
>if 8 > 9 then return 8 elseif 5 >= 4 then return 5 else return 9 end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.const  8  ; 0
.const  9  ; 1
.const  5  ; 2
.const  4  ; 3
[01] lt        0   257 256  ; 9 8, to [3] if true    (9 < 8)
[02] jmp       3           ; to [6]
[03] loadk     0   0       ; 8                   (1st true block)
[04] return    0   2
[05] jmp       7           ; to [13]
[06] le        0   259 258  ; 4 5, to [8] if true    (4 <= 5)
[07] jmp       3           ; to [11]
[08] loadk     0   2       ; 5                   (2nd true block)
[09] return    0   2
[10] jmp       2           ; to [13]
[11] loadk     0   1       ; 9                   (2nd false block)
[12] return    0   2
[13] return    0   1
; end of function
```

This example is a little more complex, but the blocks are structured in the same order as the Lua source, so interpreting the disassembled code should not be too hard.

这个例子稍微复杂些，但是程序块按照和 Lua 源码一样的顺序组织，所以解释反汇编编码应该不难。

Next are the two instructions used for performing boolean tests and implementing Lua's logic operators:

接下来是两条执行布尔测试和实现 Lua 的逻辑操作符的指令：

TEST	**A C**	if not (R(A) <=> C) then PC++
TESTSET	**A B C**	if (R(B) <=> C) then R(A) := R(B) else PC++

Used to implement **and** and **or** logical operators, or for testing a single register in a conditional statement.

用于实现逻辑操作符 **and** 和 **or** 或测试条件语句中的一个寄存器。

For TESTSET, register R(B) is coerced into a boolean and compared to the boolean field C. If R(B) matches C, the next instruction is skipped, otherwise R(B) is assigned to R(A) and the VM continues with the next instruction. The **and** operator uses a C of 0 (false) while **or** uses a C value of 1 (true).
对 TESTSET，寄存器 R(B)被强制转为布尔值并与布尔字段 C 比较。如果 R(B)与 C 匹配则跳过下一条指令，否则把 R(B)赋给 R(A)且 VM 继续执行下一条指令。（译注－这句似乎说反了。）操作符 **and** 的 C 为 0（false）而 **or** 的 C 值为 1（true）。（译注－不一定，比如 a = not b and c，字段 C 为 1，a = not b or c，字段 C 为 0。）

TEST is a more primitive version of TESTSET. TEST is used when the assignment operation is not needed, otherwise it is the same as TESTSET except that the operand slots are different.
TEST 是更原始版本的 TESTSET。当需要赋值操作时使用 TEST，此外除了操作数位置不同其他都一样。

For the fall-through case, a JMP is always expected, in order to optimize execution in the virtual machine. In effect, TEST and TESTSET must always be paired with a following JMP instruction.
为了优化虚拟机中的执行绪，失败的情况总是期望一个 JMP。实际上，TEST 和 TESTSET 必须总是后跟一个 JMP 指令，成对出现。

TEST and TESTSET are used in conjunction with a following JMP instruction, while TESTSET has an addditional conditional assignment. Like EQ, LT and LE, the following JMP instruction is compulsory, as the virtual machine will execute the JMP together with TEST or TESTSET. The two instructions are used to implement short-circuit LISP-style logical operators that retains and propagates operand values instead of booleans. First, we'll look at how **and** and **or** behaves:

TEST 和 TESTSET 与后续的 JMP 指令联合使用，而 TESTSET 有额外的条件赋值。与 EQ、LT 和 LE 一样，后续的 JMP 指令是必需的，因为虚拟机将把 JMP 与 TEST 或 TESTSET 一起执行。这两条指令用来实现 LISP 风格的短路逻辑操作符，保留和传播操作数的值而非布尔值。首先我们来看 **and** 和 **or** 的行为表现如何：

```
>local a,b,c; c = a and b
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
[1] testset    2   0   0    ; to [3] if true
[2] jmp       1           ; to [4]
[3] move      2   1
[4] return     0   1
; end of function
```

An **and** sequence exits on *false operands* (which can be **false** or **nil**) because any **false**

operands in a string of **and** operations will make the whole boolean expression **false**. If operands evaluates to **true**, evaluation continues. When a string of **and** operations evaluates to true, the result is the *last* operand value.

And 序列遇到 *false 操作数*时（可为 **false** 或 **nil**）退出，因为一串 **and** 操作中的任何 **false** 操作数将使整个布尔表达式为 **false**。如果操作数求值为 **true**，则求值继续进行。当一串 **and** 求值为 true 时，结果是*最后*一个操作数的值。

In line [1], the first operand (the local **a**) is set to local **c** when the test is **false** (with a field C of 0), while the jump to [3] is made when the test is **true**, and then in line [3], the expression result is set to the second operand (the local **b**). This is equivalent to:

在行[1]中，当测试为 **false**（字段 C 为 0）时第一个操作数（局部变量 **a**）被设为局部变量 **c**，而当测试为 **true** 时跳到[3]，然后在行[3]中，表达式的结果被设为第二个操作数（局部变量 **b**）。这等价于：

```
if a then
  c = b     -- executed by MOVE on line [3] 由行[3]上的 MOVE 执行
else
  c = a     -- executed by TESTSET on line [1] 由行[1]上的 TESTSET 执行
end
```

The **c = a** portion is done by TESTSET itself, while MOVE performs **c = b**. Now, if the result is already set with one of the possible values, a TEST instruction is used instead:

c = a 部分是 TESTSET 自身做的，而 MOVE 执行了 **c = b**。现在，如果已经用一个可能的值设置了结果，则用 TEST 指令代替：

```
>local a,b; a = a and b
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.local  "b"  ; 1
[1] test      0      0    ; to [3] if true
[2] jmp       1           ; to [4]
[3] move      0   1
[4] return    0   1
; end of function
```

The TEST instruction does not perform an assignment operation, since **a = a** is redundant. This makes TEST a little faster. This is equivalent to:

TEST 指令不执行赋值操作，因为 **a = a** 是多余的。这使 TEST 稍微快一些。这等价于：

```
if a then
  a = b
end
```

Next, we will look at the **or** operator:

我们接着看 **or** 操作符：

```
>local a,b,c; c = a or b
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
[1] testset    2   0   1    ; to [3] if false
[2] jmp       1          ; to [4]
[3] move      2   1
[4] return     0   1
; end of function
```

An **or** sequence exits on *true operands*, because any operands evaluating to **true** in a string of **or** operations will make the whole boolean expression **true**. If operands evaluates to **false**, evaluation continues. When a string of **or** operations evaluates to **false**, all operands must have evaluated to **false**.

or 序列遇到 *true 操作数*时退出，因为一串 **or** 操作中的任何操作数求值为 **true** 将使整个布尔表达式为 **true**。如果操作数求值为 **false** 则求值继续进行。当一串 **or** 操作求值为 **false** 时，所有操作数必须求值为 **false**。

In line [1], the local **a** value is set to local **c** if it is **true**, while the jump is made if it is **false** (the field C is 1). Thus in line [3], the local **b** value is the result of the expression if local **a** evaluates to **false**. This is equivalent to:

在行[1]中，如果局部变量 **c** 为 **true** 则用它设置局部变量 **a** 的值，而如果它为 **false**（字段 C 为 1）则跳转。因此在行[3]中，如果局部变量 **a** 求值为 false 则局部变量 **b** 的值是表达式的结果。这等价于：

```
if a then
  c = a     -- executed by TESTSET on line [1] 在行[1]由 TESTSET 执行
else
  c = b     -- executed by MOVE on line [3] 在行[3]由 MOVE 执行
end
```

Like the case of **and**, TEST is used when the result already has one of the possible values, saving an assignment operation:

与 **and** 的情况一样，当结果已经有一个可能的值时使用 TEST，节省了一个赋值操作：

```
>local a,b; a = a or b
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.local  "b"  ; 1
[1] test      0      1    ; to [3] if false
[2] jmp       1          ; to [4]
[3] move      0   1
[4] return     0   1
; end of function
```

Short-circuit logical operators also means that the following Lua code does not require the use of a boolean operation:

短路逻辑操作符也意味着下面的 Lua 代码并不要使用布尔操作：

```
>local a,b,c; if a > b and a > c then return a end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
[1] lt        0   1   0    ; to [3] if true
[2] jmp       3          ; to [6]
[3] lt        0   2   0    ; to [5] if true
[4] jmp       1          ; to [6]
[5] return     0   2
[6] return     0   1
; end of function
```

With short-circuit evaluation, **a > c** is never executed if **a > b** is **false**, so the logic of the Lua statement can be readily implemented using the normal conditional structure. If both **a > b** and **a > c** are **true**, the path followed is [1] (the **a > b** test) to [3] (the **a > c** test) and finally to [5], returning the value of **a**. A TEST instruction is not required. This is equivalent to:

借助短路求值，如果 **a > b** 为 **false** 则 **a > c** 永不被执行，因此可用一般的条件结构容易地实现 Lua 语句的逻辑。如果 **a > b** 和 **a > c** 都为 **true**，依照的路径是[1]（**a > b** 测试）到[3]（**a > c** 测试）并最后到[5]，返回 **a** 的值。没有要求 TEST 指令。这等价于：

```
if a > b then
  if a > c then
    return a
  end
end
```

For a single variable used in the expression part of a conditional statement, TEST is used to boolean-test the variable:

对于用在条件语句的表达式部分的一个变量，TEST 被用于变量的布尔测试：

```
>if Done then return end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.const  "Done"  ; 0
[1] getglobal  0   0       ; Done
[2] test      0      0    ; to [4] if true
[3] jmp       1          ; to [5]
[4] return     0   1
[5] return     0   1
; end of function
```

In line [2], the TEST instruction jumps to the true block if the value in temporary register 0 (from the global **Done**) is **true**. The JMP at line [3] jumps over the true block, which is the code inside the **if** block (line [4].)

在行[2]中，如果临时的 0 号寄存器中的值（赖在全局变量 **Done**）是 **true** 则 TEST 指令跳到 true 块。行[3]的 JMP 跳过 true 块，即 **if** 块内的代码（行[4]）。

If the test expression of a conditional statement consist of purely boolean operators, then a number of TEST instructions will be used in the usual short-circuit evaluation style:

如果条件语句的测试表达式完全由布尔操作符组成，那么将在通常的短路求值风格中使用许多 TEST 指令：

```
>if Found and Match then return end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.const  "Found"  ; 0
.const  "Match"  ; 1
[1] getglobal  0   0       ; Found
[2] test       0       0   ; to [4] if true
[3] jmp        4           ; to [8]
[4] getglobal  0   1       ; Match
[5] test       0       0   ; to [7] if true
[6] jmp        1           ; to [8]
[7] return     0   1
[8] return     0   1
; end of function
```

In the last example, the true block of the conditional statement is executed only if both **Found** and **Match** evaluates to **true**. The path is from [2] (test for **Found**) to [4] to [5] (test for **Match**) to [7] (the true block, which is an explicit **return** statement.)

在上例中，只有当 **Found** 和 **Match** 都求值为 **true** 时，条件语句的 true 块才被执行。路径是从[2]（**Found** 测试）到[4]到[5]（**Match** 测试）到[7]（true 块，一条显式的 **return** 语句）。

If the statement has an **else** section, then the JMP on line [6] will jump to the false block (the **else** block) while an additional JMP will be added to the true block to jump over this new block of code. If **or** is used instead of **and**, the appropriate C operand will be adjusted accordingly.

如果语句有 **else** 部分，则行[6]的 JMP 将跳到 false 块（**else** 块），同时会增加额外的 JMP 到 true 块来跳过这个新代码块。如果用 **or** 代替 **and**，将会适当的 C 操作数也会相应地被调整。

Finally, here is how Lua's ternary operator (**:?** in C) equivalent works:

最后，这里是 Lua 的三元操作符（C 中的:?）等价物如何运转：

```
>local a,b,c; a = a and b or c
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
```

```
[1] test      0      0    ; to [3] if true
[2] jmp       2           ; to [5]
[3] testset    0   1   1    ; to [5] if false
[4] jmp       1           ; to [6]
[5] move      0   2
[6] return    0   1
; end of function
```

The TEST in line [1] is for the **and** operator. First, local **a** is tested in line [1]. If it is **false**, then execution continues in [2], jumping to line [5]. Line [5] assigns local **c** to the end result because since if **a** is **false**, then **a and b** is **false**, and **false or c** is **c**.

行[1]中的 TEST 用于 **and** 操作符。首先，在行[1]中测试局部变量 **a**。如果它是 **false**，则执行绪继续进行到[2]，跳到行[5]。行[5]把局部变量 **c** 赋给最终结果，因为如果 **a** 是 **false**，则 **a and b** 是 **false**，然后 **false or c** 是 **c**。

If local **a** is **true** in line [1], the TEST instruction makes a jump to line [3], where there is a TESTSET, for the **or** operator. If **b** evaluates to **true**, then the end result is assigned the value of **b**, because **b or c** is **b** if **b** is not **false**. If **b** is also **false**, the end result will be **c**.

如果行[1]中局部变量 **a** 是 **true**，则 TEST 指令跳到行[3]，那儿是个 TESTSET，用于 **or** 操作符。如果 **b** 求值为 **true**，则最终结果被赋为 **b** 的值，因为如果 **b** 不是 **false** 则 **b or c** 是 **b**。如果 **b** 也是 **false** 则最终结果将是 **c**。

For the instructions in line [1], [3] and [5], the target (in field A) is register 0, or the local **a**, which is the location where the result of the boolean expression is assigned. The equivalent Lua code is:

对于行[1]、[3]和[5]中的指令，目标是（在字段 A 中）0 号寄存器，或者说局部变量 **a**，它是布尔表达式的结果要被赋给的位置。等价的 Lua 代码是：

```
if a then
  if b then
    a = b
  else
    a = c
  end
else
  a = c
end
```

The two **a = c** assignments are actually the same piece of code, but are repeated here to avoid using a **goto** and a label. Normally, if we assume b is not **false** and not **nil**, we end up with the more recognizable form:

两个 **a = c** 赋值实际是同一片代码，在这儿重复是为了避免使用 **goto** 和标签。通常，如果我们假定 b 不是 **false** 也不是 **nil**，可以用更易辨认的形式结束：

```
if a then
  a = b    -- assuming b ~= false
else
  a = c
end
```

12 Loop Instructions 循环指令

Lua has dedicated instructions to implement the two types of **for** loops, while the other two types of loops uses traditional test-and-jump.

Lua 有专门的指令来实现两种类型的 **for** 循环，而其他两种循环使用传统的测试和跳转。

FORPREP	**A sBx**	R(A) -= R(A+2); PC += sBx

FORLOOP	**A sBx**	R(A) += R(A+2)
		if R(A) <?= R(A+1) then {
		PC += sBx; R(A+3) = R(A)
		}

FORPREP initializes a numeric **for** loop, while FORLOOP performs an iteration of a numeric **for** loop.
FORPREP 初始化数字 **for** 循环，FORLOOP 执行数字 **for** 循环的一次迭代。

A numeric for loop requires 4 registers on the stack, and each register must be a number. R(A) holds the initial value and doubles as the internal loop variable (the *internal index*); R(A+1) is the limit; R(A+2) is the stepping value; R(A+3) is the actual loop variable (the *external index*) that is local to the **for** block.
数字 for 循环要求栈上的 4 个寄存器，每个寄存器都必须是数值。R(A)持有初始值并作为内部循环变量（内部索引）；R(A+1)是界限；R(A+2)是步进值；R(A+3)是局部于 **for** 块的实际循环变量（外部索引）。

FORPREP sets up a **for** loop. Since FORLOOP is used for initial testing of the loop condition as well as conditional testing during the loop itself, FORPREP performs a *negative step* and jumps unconditionally to FORLOOP so that FORLOOP is able to correctly make the initial loop test. After this initial test, FORLOOP performs a loop step as usual, restoring the initial value of the loop index so that the first iteration can start.
FORPREP 准备 **for** 循环。由于 FORLOOP 被用于循环条件的初始测试以及循环期间的条件测试，所以 FORPREP 执行一次负的步进并无条件跳转到 FORLOOP 以便 FORLOOP 能正确地进行初始循环测试。在这次初始测试之后，FORLOOP 执行一次寻常的循环步进，回复循环索引的初值以便首次迭代能启动。

In FORLOOP, a jump is made back to the start of the loop body if the limit has not been reached or exceeded. The sense of the comparison depends on whether the stepping is negative or positive, hence the “<?=” operator. Jumps for both instructions are encoded as signed displacements in the sBx field. An empty loop has a FORLOOP sBx value of -1.
在 FORLOOP 中，如果没有达到或超过界限则跳回循环体的开始处。比较的意义依赖于步进是负还是正，因此用了“<?=”操作符。两条指令的跳转都被编码为 sBx 字段中的有符号位移。空循环的 FORLOOP 的 sBx 字段值为-1。

FORLOOP also sets R(A+3), the external loop index that is local to the loop

block. This is significant if the loop index is used as an upvalue (see below.) R(A), R(A+1) and R(A+2) are not visible to the programmer.
FORLOOP 也设置 R(A+3)，局部于循环块的外部循环索引。如果循环索引被用作 upvalue（见下面）。R(A)、R(A+1)和 R(A+2)对程序员不可见。

The loop variable ends with the last value before the limit is reached (unlike C) because it is not updated unless the jump is made. However, since loop variables are local to the loop itself, you should not be able to use it unless you cook up an implementation-specific hack.
循环变量以到达界限前的最后一个值结束（与 C 不同），因为只有跳转才会更新它。不过，由于循环变量是局部于循环本身的，你应该不能使用它，除非你编造一个特定实现的 hack。

Loop indices behave a little differently in Lua 5.1 compared to Lua 5.0.2. Consider the following, where loop index **i** is used as an upvalue in the instantiation of 10 functions:

与 Lua5.0.2 相比，Lua5.1 中的循环索引表现少有不同。考虑下面，其中循环索引 **i** 被用作 10 个函数实例中的 upvalue：

```
local a = {}
for i = 1, 10 do
  a[i] = function() return i end
end
print(a[5]())
```

Lua 5.0.2 will print out 10, while Lua 5.1 will print out 5. In Lua 5.0.2, the scope of the loop index encloses the **for** loop, resulting in the creation of a single upvalue. In Lua 5.1, the loop index is truly local to the loop, resulting in the creation of 10 separate upvalues.

Lua5.0.2 将输出 10，可是 Lua5.1 将输出 5。在 Lua5.0.2 中，循环索引的作用域包裹着 **for** 循环，导致创建一个 upvalue。在 Lua5.1 中，循环索引是真的局部于循环，导致创建 10 个独立的 upvalue。

For the sake of efficiency, FORLOOP contains a lot of functionality, so when a loop iterates, only *one* instruction, FORLOOP, is needed. Here is a simple example:

出于性能方面的考虑，FORLOOP 包含很多功能，所以当循环迭代时，只需要一条指令，FORLOOP。这是个简单的例子：

```
>local a = 0; for i = 1,100,5 do a = a + i end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 5 stacks
.function  0 0 2 5
.local  "a"  ; 0
.local  "(for index)"  ; 1
.local  "(for limit)"  ; 2
.local  "(for step)"  ; 3
.local  "i"  ; 4
.const  0  ; 0
.const  1  ; 1
.const  100  ; 2
.const  5  ; 3
```

```
[1] loadk     0   0       ; 0
[2] loadk     1   1       ; 1
[3] loadk     2   2       ; 100
[4] loadk     3   3       ; 5
[5] forprep    1   1       ; to [7]
[6] add       0   0   4
[7] forloop    1   -2      ; to [6] if loop
[8] return    0   1
; end of function
```

In the above example, notice that the **for** loop causes three additional local pseudo-variables (or internal variables) to be defined, apart from the external loop index, **i**. The three pseudo-variables, named **(for index)**, **(for limit)** and **(for step)** are required to completely specify the state of the loop, and are not visible to Lua source code. They are arranged in consecutive registers, with the external loop index given by R(A+3) or register 4 in the example.

在上例中，注意，除了外部循环索引 **i**，**for** 循环致使定义了三个额外的局部伪变量（或内部变量）。这三个伪变量，名为**(for index)**、**(for limit)**和**(for step)**用来完整地指定循环状态，并且对 Lua 源代码是不可见的。它们与外部循环索引，例子中由 R(A+3)或 4 号寄存器给出，被安排在连续的寄存器中。

The loop body is in line [6] while line [7] is the FORLOOP instruction that steps through the loop state. The sBx field of FORLOOP is negative, as it always jumps back to the beginning of the loop body.

循环体在行[6]中，行[7]是步进遍历循环状态的 FORLOOP 指令。FORLOOP 的 sBx 字段是负的，因为它总是跳回到循环体的起点。

Lines [2]–[4] initializes the three register locations where the loop state will be stored. If the loop step is not specified in the Lua source, a constant 1 is added to the constant pool and a LOADK instruction is used to initialize the pseudo-variable **(for step)** with the loop step.

行[2]-[4]初始化存储循环状态的三个寄存器位置。如果 Lua 源代码中没指定循环步长，则像常量池中加入常量 1 并用 LOADK 指令用循环步长初始化该伪变量**(for step)**。

FORPREP in lines [5] makes a negative loop step and jumps to line [7] for the initial test. In the example, at line [5], the internal loop index (at register 1) will be (1-5) or -4. When the virtual machine arrives at the FORLOOP in line [7] for the first time, one loop step is made prior to the first test, so the initial value that is actually tested against the limit is (-4+5) or 1. Since 1 < 100, an iteration will be performed. The external loop index **i** is then set to 1 and a jump is made to line [6], thus starting the first iteration of the loop.

行[5]中的 FORPREP 产生一个负的循环步长并跳到行[7]进行初始测试。在例子中的行[5]，内部循环索引（在 1 号寄存器）将为（1-5）或-4。当虚拟机首次到达行[7]的 FORLOOP 时，在首次测试前先产生一次循环步进，所以以界限为标准实际测试的初始值是（-4+5）或 1。由于 1 < 100，将执行一次迭代。接着外部循环索引 **i** 被设为 1 并跳到行[6]，由此开始循环的首次迭代。

The loop at line [6]–[7] repeats until the internal loop index exceeds the loop limit of 100.

The conditional jump is not taken when that occurs and the loop ends. Beyond the scope of the loop body, the loop state (**(for index)**, **(for limit)**, **(for step)** and **i**) is not valid. This is determined by the parser and code generator. The range of PC values for which the loop state variables are valid is located in the locals list. The brief assembly listings generated by ChunkSpy that you are seeing does not give the **startpc** and **endpc** values contained in the locals list. In theory, these rules can be broken if you write Lua assembly directly.

行[6]-[7]的循环重复直到内部索引超过循环界限 100。当那发生时不进行条件跳转且循环结束。在循环体作用域外，循环状态（**(for index)**、**(for limit)**、**(for step)**和 **i**）是无效的。这由解析器和编码生成器进行检查。（表示）循环状态变量的有效范围的 PC 值位于局部变量列表中。你所看到的 ChunkSpy 生成的概要汇编清单没有给出局部变量列表中包含的 **startpc** 和 **endpc**。理论上，如果你直接编写 Lua 汇编能打破这些规则。

```
>for i = 10,1,-1 do if i == 5 then break end end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 4 stacks
.function  0 0 2 4
.local  "(for index)"  ; 0
.local  "(for limit)"  ; 1
.local  "(for step)"  ; 2
.local  "i"  ; 3
.const  10  ; 0
.const  1  ; 1
.const  -1  ; 2
.const  5  ; 3
[1] loadk     0   0       ; 10
[2] loadk     1   1       ; 1
[3] loadk     2   2       ; -1
[4] forprep    0   3       ; to [8]
[5] eq        0   3   259  ; 5, to [7] if true
[6] jmp       1          ; to [8]
[7] jmp       1          ; to [9]
[8] forloop    0   -4      ; to [5] if loop
[9] return    0   1
; end of function
```

In the second loop example above, except for a negative loop step size, the structure of the loop is identical. The body of the loop is from line [5] to line [8]. Since no additional stacks or states are used, a **break** translates simply to a JMP instruction (line [7]). There is nothing to clean up after a FORLOOP ends or after a JMP to exit a loop.

在上面的第二个循环例子中，除了负的循环步进尺寸外，循环的结构是完全一样的。循环体从行[5]到行[8]。由于没用额外的栈或状态，**break** 只是翻译成一条 JMP 指令（行[7]）。在一条 FORLOOP 结束后或一条 JMP 退出循环后没有东西要清理。

Apart from a numeric **for** loop (implemented by FORPREP and FORLOOP), Lua has a generic **for** loop, implemented by TFORLOOP:

除了数字 **for** 循环（由 FORPREP 和 FORLOOP 实现），Lua 还有泛型 **for** 循环，由 TFORLOOP 实现：

TFORLOOP **A C**

```
R(A+3), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2));
if R(A+3) ~= nil then {
    R(A+2) = R(A+3);
} else {
    PC++;
}
```

Performs an iteration of a generic **for** loop. A Lua 5-style generic **for** loop keeps 3 items in consecutive register locations to keep track of things. R(A) is the *iterator function*, which is called once per loop. R(A+1) is the *state*, and R(A+2) is the enumeration index. At the start, R(A+2) has an initial value. R(A), R(A+1) and R(A+2) are internal to the loop and cannot be accessed by the programmer; at first, they are set with an initial state.
执行一次泛型 **for** 循环的迭代。Lua5 风格的泛型 **for** 循环保有 3 项连续的寄存器位置来跟踪状态。R(A)是*迭代函数*，每个循环调用一次。R(A+1)是*状态*，R(A+2)是枚举索引。刚开始，R(A+2)具有初值。R(A)、R(A+1)和 R(A+2)在循环内部，不能被程序员访问；它们起初被设为初始状态。

In addition to these internal loop variables, the programmer specifies one or more loop variables that are external and visible to the programmer. These loop variables reside at locations R(A+3) onwards, and their count is specified in operand C. Operand C must be at least 1. They are also local to the loop body, like the external loop index in a numerical **for** loop.
除了这些内部循环变量，程序员指定一个或多个对程序员可见的外部循环变量。这些循环变量驻留在 R(A+3)开始的位置，它们的数量由操作数 C 指定。操作数 C 必须至少为 1.它们也是局部于循环体内的，同数字 **for** 循环中的外部循环索引一样。

Each time TFORLOOP executes, the iterator function referenced by R(A) is called with two arguments: the state and the enumeration index (R(A+1) and R(A+2).) The results are returned in the local loop variables, from R(A+3) onwards, up to R(A+2+C).
TFORLOOP 每次执行时，R(A)引用的迭代器函数被调用，有两个参数：状态和枚举索引（R(A+1)和 R(A+2)）。结果返回到从 R(A+3)开始直到 R(A+2+C)的局部循环变量中。

Next, the first return value, R(A+3), is tested. If it is **nil**, the iterator loop is at an end, and TFORLOOP skips the next instruction and the **for** loop block ends. Note that the state of the generic **for** loop does not depend on any of the external iterator variables that are visible to the programmer.
接着，测试第一个返回值，R(A+3)。如果它是 nil，则迭代器循环到达末尾，且 TFORLOOP 跳过下一条指令从而 **for** 循环块终止。注意，泛型 **for** 循环的状态不依赖于任何对程序员可见的外部迭代器变量。

If R(A+3) is not **nil**, there is another iteration, and R(A+3) is assigned as the new value of the enumeration index, R(A+2). Then next instruction, which *must* be a JMP, is immediately executed, sending execution back to the beginning of the loop. This is an optimization case; TFORLOOP will not work correctly without the JMP instruction.

如果 R(A+3)不是 **nil**，则还有另一次迭代，并且 R(A+3)作为枚举索引的新值赋给 R(A+2)。然后下一条指令，它*必须*是个 JMP，立刻执行，把执行绪发送回循环的起点。这是种优化情形；没有 JMP 指令则 TFORLOOP 将不会正确运转。

Like the numerical **for** loop, the generic **for** loop behave a little differently in Lua 5.1 compared to Lua 5.0.2. In the following example:

同数字 **for** 循环一样，与 Lua5.0.2 相比，Lua5.1 中的泛型 **for** 循环行为表现也稍有不同。在下例中：

```
local a = {[1]=2,[2]=4,[3]=8}
local b = {}
for i,v in pairs(a) do
  b[i] = function() return v end
end
print(b[1](), b[2](), b[3]())
```

Lua 5.0.2 will print out 3 **nil**s, while Lua 5.1 will print out 2, 4 and 8. In Lua 5.0.2, the scope of the external iterator variables encloses the **for** loop, resulting in the creation of a single upvalue. In Lua 5.1, the iterator variables are truly local to the loop, resulting in the creation of separate upvalues.

Lua5.0.2 将输出 3 个 **nil**，而 Lua5.1 将输出 2、4 和 8。在 Lua5.0.2 中，外部迭代器变量的作用域包裹着 **for** 循环，导致创建一个 upvalue。在 Lua5.1 中，迭代器变量是真的局部于循环的，导致创建独立的 upvalue。

This example has a loop with one additional result (**v**) in addition the loop enumerator (**i**):

该例的循环除了循环枚举器（**i**）还有个额外的结果（**v**）：

```
>for i,v in pairs(t) do print(i,v) end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 8 stacks
.function  0 0 2 8
.local  "(for generator)"  ; 0
.local  "(for state)"  ; 1
.local  "(for control)"  ; 2
.local  "i"  ; 3
.local  "v"  ; 4
.const  "pairs"  ; 0
.const  "t"  ; 1
.const  "print"  ; 2
[01] getglobal  0   0       ; pairs
[02] getglobal  1   1       ; t
[03] call       0   2   4
[04] jmp        4           ; to [9]
[05] getglobal  5   2       ; print
[06] move       6   3
[07] move       7   4
[08] call       5   3   1
[09] tforloop   0       2   ; to [11] if exit
[10] jmp        -6          ; to [5]
```

```
[11] return    0   1
; end of function
```

The iterator function is located in register 0, and is named **(for generator)** for debugging purposes. The state is in register 1, and has the name **(for state)**. The enumeration index, **(for control)**, is contained in register 2. These correspond to locals R(A), R(A+1) and R(A+2) in the TFORLOOP description. Results from the iterator function call is placed into register 3 and 4, which are locals **i** and **v**, respectively. On line [9], the operand C of TFORLOOP is 2, corresponding to two iterator variables (**i** and **v**).

迭代器函数位于 0 号寄存器，并出于调试目的命名为**(for generator)**。状态在 1 号寄存器，名为**(for state)**。枚举索引**(for control)**包含在 2 号寄存器中。这些在 TFORLOOP 的描述中对应于局部变量 R(A)、R(A+1)和 R(A+2)。赖在迭代器函数调用的结果置于 3 号和 4 号寄存器中，它们分别是局部变量 **i** 和 **v**。在行[9]，TFORLOOP 的操作数 C 是 2，对应于两个迭代器变量（**i** 和 **v**）。

Line [1]–[3] prepares the iterator state. Note that the call to the **pairs** standard library function has 1 parameter and 3 results. After the call in line [3], register 0 is the iterator function, register 1 is the loop state, register 2 is the initial value of the enumeration index. The iterator variables **i** and **v** are both invalid at the moment, because we have not entered the loop yet.

行[1]-[3]准备迭代器状态。注意，对标准库函数 **pairs** 的调用具有 1 个参数和 3 个结果。在行[3]的调用之后，0 号寄存器是迭代函数，1 号寄存器是循环状态，2 好寄存器是枚举索引的初值。此时迭代变量 **i** 和 **v** 都是无效的，因为我们还未进入循环。

Line [4] is a JMP to TFORLOOP on line [9]. With the initial (or *zeroth*) iterator state, TFORLOOP calls the iterator function, generating the first set of enumeration results in locals **i**, **v**. If **i** is not **nil**, the internal enumeration index (register 2) is set and the JMP on the next line is immediately executed, starting the first iteration of the loop body (lines [5]–[8]).

行[4]的 JMP 到行[9]的 TFORLOOP。藉着初始（或第 0 个）迭代器状态，TFORLOOP 调用迭代器函数生成局部变量 **i**、**v** 中的第一组枚举结果。如果 **i** 不是 **nil**，那么内部枚举索引（2 号寄存器）被设置且下一行的 JMP 立刻执行，开始了循环体（行[5]-[8])的第一次迭代。

The body of the generic **for** loop executes (`print(i,v)`) and then TFORLOOP is encountered again, calling the iterator function to get the next iteration state. Finally, when the first result is a **nil**, the loop ends, and execution continues on line [11].

泛型 **for** 的循环体执行（`print(i,v)`）然后再次遇到 TFORLOOP，调用迭代器函数以的奥下一个迭代状态。最后，当第一个结果为 **nil** 时，循环结束，并且执行绪继续执行到行[11]。

repeat and **while** loops use a standard test-and-jump structure. Here is a **repeat** loop:

repeat 和 **while** 循环使用标准的测试和跳转结构。这是 **repeat** 循环：

```
>local a = 0; repeat a = a + 1 until a == 10
```

```
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.const  0  ; 0
.const  1  ; 1
.const  10  ; 2
[1] loadk     0   0       ; 0
[2] add       0   0   257  ; 1
[3] eq        0   0   258  ; 10, to [5] if true
[4] jmp       -3          ; to [2]
[5] return    0   1
; end of function
```

The body of the **repeat** loop is line [2], while the test-and-jump scheme is implemented in lines [3] and [4]. Although two instructions are needed to loop the loop, Lua 5.1 executes EQ and JMP together, saving some time.

repeat 的循环体是行[2]，测试和跳转方案在行[3]和[4]中实现。尽管需要两条指令来处理循环，Lua5.1 把 EQ 和 JMP 一起执行，节省了一些时间。

```
>local a = 1; while a < 10 do a = a + 1 end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.const  1  ; 0
.const  10  ; 1
[1] loadk     0   0       ; 1
[2] lt        0   0   257  ; 10, to [4] if true
[3] jmp       2           ; to [6]
[4] add       0   0   256  ; 1
[5] jmp       -4          ; to [2]
[6] return    0   1
; end of function
```

For a **while** loop, the test (line[2]) is made first. If the test is **true**, execution continues with the loop body (line [4]). A JMP on line [5] returns execution to the loop test instruction. This is a little different from Lua 5.0.2 **while** loops, which have the loop test at the end of the loop block and has a loop condition size limitation.

对 **while** 循环来说，测试（行[2]）是先做的。如果测试为 **true** 则执行绪继续执行循环体（行[4]）。行[5]的 JMP 把执行绪返回到循环测试指令。这与 Lua5.0.2 的 **while** 循环有些不同，它的循环测试在循环块的末尾，并且具有循环条件的尺寸限制。

A **while** loop in the Lua 5.0.2 style will look like this:

Lua5.0.2 中的 **while** 循环看起来像这样：

```
>local a = 1; while a < 10 do a = a + 1 end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "a"  ; 0
.const  1  ; 0
```

```
.const  10  ; 1
[1] loadk     0   0       ; 1
[2] jmp       1           ; to [4]
[3] add       0   0   256  ; 1
[4] lt        1   0   257  ; 10, to [6] if false
[5] jmp       -3          ; to [3]
[6] return    0   1
; end of function
```

The sense of the condition test is reversed, while the loop body is at line [3]. The condition test is made at the end of the loop on line [4].

条件测试的意义反转了，而循环体在行[3]。条件测试在循环末尾的行[4]上进行。

13 Table Creation 表创建

There are two instructions for table creation and initialization. One instruction creates a table while the other instruction sets the array elements of a table.

有两条指令用于表创建和初始化。一条指令创建表而另一条指令设置表的数组部分。

NEWTABLE A B C R(A) := {} (size = B,C)

Creates a new empty table at register R(A). B and C are the encoded size information for the array part and the hash part of the table, respectively. Appropriate values for B and C are set in order to avoid rehashing when initially populating the table with array values or hash key-value pairs.
在寄存器 R(A)处创建新的空表。B 和 C 分别用于表的数组部分和散列部分的编码后的尺寸信息。当用数组值或散列键-值对初始填充表时，用合适的值设置 B 和 C 以避免再散列。

Operand B and C are both encoded as a "floating point byte" (so named in `lobject.c`) which is `eeeeexxx` in binary, where `x` is the mantissa and `e` is the exponent. The actual value is calculated as `1xxx*2^(eeeee-1)` if `eeeee` is greater than 0 (a range of 8 to 15*2^30.) If `eeeee` is 0, the actual value is `xxx` (a range of 0 to 7.)
操作数 B 和 C 都被编码为"浮点字节"（在 `lobject.c` 中如此命名），它的二进制形式是 `eeeeexxx`，其中 `x` 是尾数 `e` 是指数。如果 `eeeee` 大于 0（范围 8 到 15*2^30)则实际值计算为 `1xxx*2^(eeeee-1)`。如果 `eeeee` 是 0 则实际值是 `xxx`（范围 0 到 7）。

If an empty table is created, both sizes are zero. If a table is created with a number of objects, the code generator counts the number of array elements and the number of hash elements. Then, each size value is rounded up and encoded in B and C using the floating point byte format.
如果创建的是空表，两个尺寸都是 0.如果带很多对象创建表，则编码生成器统计数组元素的数量和散列元素的数量。然后把每个尺寸都向上舍入并用浮点字节格式编码到 B 和 C 中。

Creating an empty table forces both array and hash sizes to be zero:

创建空表会强制数组和散列尺寸为 0：

```
>local q = {}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "q"  ; 0
[1] newtable   0   0   0    ; array=0, hash=0
[2] return     0   1
; end of function
```

In later examples, we will see how the size values are encoded. But first, we need to learn about the SETLIST instruction, which is used to initialize array elements in a table.

在稍后的例子中，我们将看看尺寸值是如何编码的。但是首先，我们需要学习 SETLIST 指令，它用于初始化表的数组元素。

SETLIST **A B C** R(A)[(C-1)*FPF+i] := R(A+i), 1 <= i <= B

Sets the values for a range of *array elements* in a table referenced by R(A). Field B is the number of elements to set. Field C encodes the block number of the table to be initialized. The values used to initialize the table are located in registers R(A+1), R(A+2), and so on.
把一系列*数组元素*设置到 R(A)引用的表中。字段 B 是要设置的元素数。字段 C 编码了要初始化的表的块编号。用来初始化表的值位于寄存器 R(A+1)、R(A+2)，等等。

The block size is denoted by FPF. FPF is “fields per flush”, defined as LFIELDS_PER_FLUSH in the source file `lopcodes.h`, with a value of 50. For example, for array locations 1 to 20, C will be 1 and B will be 20.
块尺寸由 FPF 指出。FPF 是“每次刷新的字段”，在源文件 `lopcodes.h` 中定义为 LFIELDS_PER_FLUSH，值为 50。例如，对于数组位置 1 到 20，C 将为 1B 将为 20。

If B is 0, the table is set with a variable number of array elements, from register R(A+1) up to the top of the stack. This happens when the last element in the table constructor is a function call or a vararg operator.
如果 B 是 0，则是用可变数量的数组元素设置表，从寄存器 R(A+1)直到栈顶。当表构造器中的最后一个元素是函数调用或 vararg 操作符时出现这种情况。

If C is 0, the *next instruction* is cast as an integer, and used as the C value. This happens only when operand C is unable to encode the block number, i.e. when C > 511, equivalent to an array index greater than 25550.
如果 C 是 0，下一条指令被转换为整数并用作 C 的值。只有当操作数 C 不能编码块编号时出现该情况，例如当 C > 511 时，等同于数组索引大于 25550。

We’ll start with a simple example:

我们以简单的例子开始：

```
>local q = {1,2,3,4,5,}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 6 stacks
.function  0 0 2 6
.local  "q"  ; 0
.const  1  ; 0
.const  2  ; 1
.const  3  ; 2
.const  4  ; 3
.const  5  ; 4
[1] newtable   0   5   0    ; array=5, hash=0
[2] loadk      1   0       ; 1
[3] loadk      2   1       ; 2
[4] loadk      3   2       ; 3
```

```
[5] loadk     4   3       ; 4
[6] loadk     5   4       ; 5
[7] setlist    0   5   1    ; index 1 to 5
[8] return    0   1
; end of function
```

A table with the reference in register 0 is created in line [1] by NEWTABLE. Since we are creating a table with no hash elements, the array part of the table has a size of 5, while the hash part has a size of 0.

在行[1]中 NEWTABLE 创建了一个表，在 0 号寄存器中引用它。由于我们要创建不带散列元素的表，所以表的数组部分尺寸为 5，而散列部分尺寸为 0。

Constants are then loaded into temporary registers 1 to 5 (lines [2] to [6]) before the SETLIST instruction in line [7] assigns each value to consecutive table elements. The start of the block is encoded as 1 in operand C. The starting index is calculated as (1-1)*50+1 or 1. Since B is 5, the range of the array elements to be set becomes 1 to 5, while the objects used to set the array elements will be R(1) through R(5).

接着在行[7]中的 SETLIST 指令把每个值赋给连续的表元素之前，常量被载入临时的寄存器 1 到 5 中（行[2]到[6]）。块的起点在操作数 C 中编码为 1。起始索引计算为 (1-1)*50+1 或 1。由于 B 是 5，要设置的数组元素的范围变为 1 到 5，而用于设置数组元素的对象将是 R(1)到 R(5)。

Next is a larger table with 55 array elements. This will require two blocks to initialize. Some lines have been removed and ellipsis (...) added to save space.

接下来是带有 55 个数组元素的更大的表。这将需要初始化两个块。移除了一些行且加入省略号（...）以节省空间。

```
>local q = {1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0, \
>>1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0, \
>>1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 51 stacks
.function  0 0 2 51
.local  "q"  ; 0
.const  1  ; 0
.const  2  ; 1
...
.const  0  ; 9
[01] newtable   0   30  0    ; array=56, hash=0
[02] loadk     1   0       ; 1
[03] loadk     2   1       ; 2
...
[51] loadk     50  9       ; 0
[52] setlist    0   50  1    ; index 1 to 50
[53] loadk     1   0       ; 1
[54] loadk     2   1       ; 2
...
[57] loadk     5   4       ; 5
[58] setlist    0   5   2    ; index 51 to 55
[59] return    0   1
; end of function
```

Since FPF is 50, the array will be initialized in two blocks. The first block is for index 1 to 50, while the second block is for index 51 to 55. Each array block to be initialized requires one SETLIST instruction. On line [1], NEWTABLE has a field B value of 30, or 00011110 in binary. From the description of NEWTABLE, `xxx` is 110_2, while `eeeee` is 11_2. Thus, the size of the array portion of the table is (1110)*2^(11-1) or (14*2^2) or 56.

由于 FPF 是 50，数组将被初始化在两块中。第一块用于索引 1 到 50，第二块用于 51 到 55。每个要初始化的数组块需要一个 SETLIST 指令。在行[1]，NEWTABLLE 的字段 B 的值为 30，或二进制的 00011110。从 NEWTABLE 的说明（可知），`xxx` 是 110_2，而 `eeeee` 是 11_2。因此表的数组部分尺寸是(1110)*2^(11-1)或(14*2^2) or 56。

Lines [2] to [51] sets the values used to initialize the first block. On line [52], SETLIST has a B value of 50 and a C value of 1. So the block is from 1 to 50. Source registers are from R(1) to R(50). Lines [53] to [57] sets the values used to initialize the second block. On line [58], SETLIST has a B value of 5 and a C value of 2. So the block is from 51 to 55. The start of the block is calculated as (2-1)*50+1 or 51. Source registers are from R(1) to R(5).

行[2]到[5]设置用来初始化第一块的值。在行[52]，SETLIST 的 B 值为 50，其 C 值为 1。所以该块石从 1 到 50。源寄存器是从 R(1)到 R(50)。行[53]到[57]设置用来初始化第二块的值。在行[58]，SETLIST 的 B 值为 5，其 C 值为 2。所以该块从 51 到 55。块的起点计算为(2-1)*50+1 或 51。源寄存器是从 R(1)到 R(5)。

Here is a table with hashed elements:

这是带有散列元素的表：

```
>local q = {a=1,b=2,c=3,d=4,e=5,f=6,g=7,h=8,}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "q"  ; 0
.const  "a"  ; 0
.const  1  ; 1
.const  "b"  ; 2
.const  2  ; 3
.const  "c"  ; 4
.const  3  ; 5
.const  "d"  ; 6
.const  4  ; 7
.const  "e"  ; 8
.const  5  ; 9
.const  "f"  ; 10
.const  6  ; 11
.const  "g"  ; 12
.const  7  ; 13
.const  "h"  ; 14
.const  8  ; 15
[01] newtable   0   0   8    ; array=0, hash=8
[02] settable   0   256 257  ; "a" 1
[03] settable   0   258 259  ; "b" 2
[04] settable   0   260 261  ; "c" 3
[05] settable   0   262 263  ; "d" 4
[06] settable   0   264 265  ; "e" 5
```

```
[07] settable   0   266 267  ; "f" 6
[08] settable   0   268 269  ; "g" 7
[09] settable   0   270 271  ; "h" 8
[10] return     0   1
; end of function
```

In line [1], NEWTABLE is executed with an array part size of 0 and a hash part size of 8. On lines [2] to line [9], key-value pairs are set using SETTABLE. The SETLIST instruction is only for initializing array elements. Using SETTABLE to initialize the key-value pairs of a table in the above example is quite efficient as it can reference the constant pool directly.

在行[1]中，NEWTABLE 以数组部分尺寸为 0 且散列部分尺寸为 8 执行。在行[2]到行[9]，使用 SETTABLE 设置键值-对。SETLIST 只用于初始化数组部分。上例中使用 SETTABLE 初始化键-值对很高效，因为它能直接引用常量池。

If there are both array elements and hash elements in a table constructor, both SETTABLE and SETLIST will be used to initialize the table after the initial NEWTABLE. In addition, if the last element of the table constructor is a function call or a vararg operator, then the B operand of SETLIST will be 0, to allow objects from R(A+1) up to the top of the stack to be initialized as array elements of the table.

如果表构造器中数组元素和散列元素都有，则在 NEWTABLE 以后将用 SETTABLE 和 SETLIST 初始化表。另外，如果表构造器的最后一个元素是函数调用或 vararg 操作符，则 SETLIST 的 B 操作数将为 0 以允许从 R(A+1)开始直到栈顶的对象都初始化为表的数组元素。

```
>return {1,2,3,a=1,b=2,c=3,foo()}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 5 stacks
.function  0 0 2 5
.const  1  ; 0
.const  2  ; 1
.const  3  ; 2
.const  "a"  ; 3
.const  "b"  ; 4
.const  "c"  ; 5
.const  "foo"  ; 6
[01] newtable   0   3   3    ; array=3, hash=3
[02] loadk      1   0       ; 1
[03] loadk      2   1       ; 2
[04] loadk      3   2       ; 3
[05] settable   0   259 256  ; "a" 1
[06] settable   0   260 257  ; "b" 2
[07] settable   0   261 258  ; "c" 3
[08] getglobal  4   6       ; foo
[09] call       4   1   0
[10] setlist    0   0   1    ; index 1 to top
[11] return     0   2
[12] return     0   1
; end of function
```

In the above example, the table is first created in line [1] with its reference in register 0, and it has both array and hash elements to be set. The size of the array part is 3 while the size of the hash part is also 3.

在上例中，首先在行[1]中创建在 0 号寄存器中引用的表，它的数组和散列元素都要设置。数组部分的尺寸是 3，散列部分的尺寸也是 3。

Lines [2]–[4] loads the values for the first 3 array elements. Lines [5]–[7] sets the 3 key-value pairs for the hash part of the table. In lines [8] and [9], the call to function **foo** is made, and then in line [10], the SETLIST instruction sets the first 3 array elements (in registers 1 to 3) plus whatever additional results returned by the **foo** function call (from register 4 onwards.) This is accomplished by setting operand B in SETLIST to 0. For the first block, operand C is 1 as usual. If no results are returned by the function, the top of stack is at register 3 and only the 3 constant array elements in the table are set.

行[2]-[4]载入最初的 3 个数组元素的值。行[5]-[7]设置表的散列部分的 3 对键-值对。在行[8]和[9]中调用函数 **foo**，然后在行[10]中 SETLIST 指令设置最初的 3 个数组元素（在 1 到 3 号寄存器中）外加任何 **foo** 函数调用返回的额外结果（从 4 号寄存器开始）。这通过设置 SETLIST 中的操作数 B 为 0 完成。对第一块而言，操作数 C 是和往常一样的 1。如果函数没返回结果，则栈顶在 3 号寄存器，且只设置了表中的 3 个常量数组元素。

```
>local a; return {a(), a(), a()}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 5 stacks
.function  0 0 2 5
.local  "a"  ; 0
[01] newtable   1   2   0    ; array=2, hash=0
[02] move       2   0
[03] call       2   1   2
[04] move       3   0
[05] call       3   1   2
[06] move       4   0
[07] call       4   1   0
[08] setlist    1   0   1    ; index 1 to top
[09] return     1   2
[10] return     0   1
; end of function
```

Note that only the last function call in a table constructor retains all results. Other function calls in the table constructor keep only one result. This is shown in the above example. For vararg operators in table constructors, please see the discussion for the VARARG instruction for an example.

注意，只有表构造器中的最后一个函数调用保留所有结果。表构造器中的其他函数调用值保留一个结果。这在上例中展示。对表构造器中的 vararg 操作符，请看 VARARG 指令的例子的讨论。

14 Closures and Closing 创建和结束闭包

The final two instructions of the Lua virtual machine are a little involved because of the handling of upvalues. The first is CLOSURE, for instantiating function prototypes:

因为 upvalue 处理的关系，Lua 虚拟机的最后两条指令有点复杂。第一个是 CLOSURE，用于实例化函数原型：

CLOSURE **A Bx** R(A) := closure(KPROTO[Bx], R(A), ... ,R(A+n))

Creates an instance (or closure) of a function. Bx is the function number of the function to be instantiated in the table of function prototypes. This table is located after the constant table for each function in a binary chunk. The first function prototype is numbered 0. Register R(A) is assigned the reference to the instantiated function object.
创建函数的一个实例（或闭包）。Bx 是要实例化的函数在函数原型表中的函数编号。在二进制程序块中，该表位于每个函数的常量表后面。第一个函数原型是编号为 0.寄存器 R(A)被赋值为被实例化的函数对象的引用。

For each upvalue used by the instance of the function KPROTO[Bx], there is a pseudo-instruction that follows CLOSURE. Each upvalue corresponds to either a MOVE or a GETUPVAL pseudo-instruction. Only the B field on either of these pseudo-instructions are significant.
对于函数 KPROTO[Bx]的实例用到的每个 upvalue，都有一条伪指令跟在 CLOSURE 后面。每个 upvalue 对应一个 MOVE 或 GETUPVAL 伪指令。这些伪指令中的每个都只有的 B 字段是有意义的。

A MOVE corresponds to local variable R(B) in the current lexical block, which will be used as an upvalue in the instantiated function. A GETUPVAL corresponds upvalue number B in the current lexical block. The VM uses these pseudo-instructions to manage upvalues.
MOVE 对应于当前词法块中的局部变量 R(B)，它将在实例化的函数中被用作 upvalue。GETUPVAL 对应于当前词法块中的编号为 B 的 upvalue。VM 使用这些伪指令管理 upvalue。

If the function prototype has no upvalues, then CLOSURE is pretty straightforward: Bx has the function number and R(A) is assigned the reference to the instantiated function object. However, when an upvalue comes into the picture, we have to look a little more carefully:

如果函数原型没有 upvalue 则 CLOSURE 相当简单：Bx 有函数编号，R(A)被赋以实例化的函数对象的引用。然而，当 upvalue 牵连进来时，我们不得不多加注意：

```
>local u; \
>>function p() return u end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "u"  ; 0
.const  "p"  ; 0
```

```
  ; function [0] definition (level 2)
  ; 1 upvalues, 0 params, 2 stacks
  .function  1 0 0 2
  .upvalue  "u"  ; 0
  [1] getupval   0   0       ; u
  [2] return     0   2
  [3] return     0   1
  ; end of function

[1] closure    1   0       ; 1 upvalues
[2] move       0   0
[3] setglobal  1   0       ; p
[4] return     0   1
; end of function
```

In the example, the upvalue in the level 2 function is **u**, and within the main chunk there is a single function prototype (indented in the listing above for clarity.) In the top-level function, line [1], the closure is made. In line [3] the function reference is saved into global **p**. Line [2] is a part of the CLOSURE instruction (it not really an actual MOVE,) and its B field specifies that upvalue number 0 in the closed function is really local **u** in the enclosing function.

例中第二层函数中的 upvalue 是 **u**，并且在主程序块中仅有一个函数原型（上面列表中为了清晰缩进了）。在顶层函数中，行[1]产生一个闭包。在行[3]中函数引用存入全局变量 **p**。行[2]是 CLOSURE 指令的一部分（它不是真的 MOVE）。并且其 B 字段指定了关闭的函数中的 upvalue 编号 0 实际上市封闭函数中的局部变量 **u**。

Here is another example, with 3 levels of function prototypes:

这是另一个例子，带 3 层函数原型：

```
>local m \
>>function p() \
>>  local n \
>>  function q() return m,n end \
>>end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "m"  ; 0
.const  "p"  ; 0

  ; function [0] definition (level 2)
  ; 1 upvalues, 0 params, 2 stacks
  .function  1 0 0 2
  .local  "n"  ; 0
  .upvalue  "m"  ; 0
  .const  "q"  ; 0

    ; function [0] definition (level 3)
    ; 2 upvalues, 0 params, 2 stacks
    .function  2 0 0 2
    .upvalue  "m"  ; 0
    .upvalue  "n"  ; 1
    [1] getupval   0   0       ; m
    [2] getupval   1   1       ; n
    [3] return     0   3
```

```
    [4] return    0   1
    ; end of function

  [1] closure   1   0       ; 2 upvalues
  [2] getupval  0   0       ; m
  [3] move      0   0
  [4] setglobal 1   0       ; q
  [5] return    0   1
  ; end of function

[1] closure   1   0       ; 1 upvalues
[2] move      0   0
[3] setglobal 1   0       ; p
[4] return    0   1
; end of function
```

First, look at the top-level function and the level 2 function – there is one upvalue, **m**. In the top-level function, the closure in line [1] has one more instruction following it (the MOVE), for the upvalue **m**. This is similar to the previous example.

先看顶层函数和 2 层函数－有个 upvalue，**m**。在顶层函数中，行[1]中的闭包后跟着另一条指令（MOVE），用于 upvalue **m**。这与前例类似。

Next, compare the level 2 function and the level 3 function – now there are two upvalues, **m** and **n**. The **m** upvalue is found 2 levels up. In the level 2 function, the closure in line [1] has two instructions following it. The first is for upvalue number 0 (**m**) – it uses GETUPVAL to indicate that the upvalue is one or more level lower down. The second is for upvalue number 1 (**n**) – it uses MOVE which indicate that the upvalue is in the same level as the CLOSURE instruction. For both of these pseudo-instructions, the B field is used to point either to the upvalue or local in question. The Lua virtual machine uses this information (CLOSURE information and upvalue lists) to manage upvalues; for the programmer, upvalues just works.

接下来比较 2 层函数和 3 层函数－现在有两个 upvalue，**m** 和 **n**。**m** upvalue 在 2 层之上找到。在 2 层函数中，行[1]中的闭包后跟两条指令。第一个用于 0 号 upvalue（**m**）－它用 GETUPVAL 来指示该 upvalue 要低一或更多层。第二个用于 1 号 upvalue（**n**）－它用 MOVE 指示该 upvalue 与 CLOSURE 指令在同一层。这两个伪指令都用 B 字段指向所关注的 upvalue 或局部变量。Lua 虚拟机用该信息（CLOSURE 信息和 upvalue 表）来管理 upvalue；对程序员来说，upvalue 能运转即可。

The last instruction to be covered in this guide, CLOSE, also deals with upvalues:

本指南涉及的最后一条指令，CLOSE，也处理 upvalue：

CLOSE **A** close all variables in the stack up to (>=) R(A)
关闭栈中直到 R(A)的所有变量

Closes all local variables in the stack from register R(A) onwards. This instruction is only generated if there is an upvalue present within those local variables. It has no effect if a local isn't used as an upvalue.
关闭栈中从 R(A)开始的所有局部变量。该指令只在那些局部变量中存在 upvalue 时生成。如果局部变量不是用作 upvalue 则它没效果。

If a local is used as an upvalue, then the local variable need to be placed somewhere, otherwise it will go out of scope and disappear when a lexical block enclosing the local variable ends. CLOSE performs this operation for all affected local variables for **do end** blocks or loop blocks. RETURN also does an implicit CLOSE when a function returns.
如果局部变量被用作 upvalue，则该局部变量需要被置于某处，否则，当词法块封闭局部变量末端时，它将超出作用域并消失。CLOSE 为 **do end** 块或循环块的所有受影响的局部变量执行此操作。当函数返回时 RETURN 也做一个隐式的 CLOSE。

It is easier to understand CLOSE with an example:

通过例子更容易理解 CLOSE：

```
>do \
>>  local p,q \
>>  r = function() return p,q end \
>>end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 2 3
.local  "p"  ; 0
.local  "q"  ; 1
.const  "r"  ; 0

  ; function [0] definition (level 2)
  ; 2 upvalues, 0 params, 2 stacks
  .function  2 0 0 2
  .upvalue  "p"  ; 0
  .upvalue  "q"  ; 1
  [1] getupval   0   0       ; p
  [2] getupval   1   1       ; q
  [3] return     0   3
  [4] return     0   1
  ; end of function

[1] closure    2   0       ; 2 upvalues
[2] move       0   0
[3] move       0   1
[4] setglobal  2   0       ; r
[5] close      0
[6] return     0   1
; end of function
```

p and **q** are local to the **do end** block, and they are upvalues as well. The global **r** is assigned an anonymous function that has **p** and **q** as upvalues. When **p** and **q** go out of scope at the end of the **do end** block, both variables have to be put somewhere because they are part of the environment of the function instantiated in **r**. This is where the CLOSE instruction comes in.

P 和 **q** 局部于 **do end** 块，它们也是 upvalue。全局变量 **r** 被赋值为以 **p** 和 **q** 为 upvalue 的匿名函数。当 **p** 和 **q** 在 **do end** 块末尾超出作用域时，两个变量不得不被放在某处，因为它们是 **r** 中的函数实例的环境的一部分。这就是 CLOSE 指令出现的地方。

In the top-level function, the CLOSE in line [5] makes the virtual machine find all affected locals (they have to be open upvalues,) take them out of the stack, and place them in a safe place so that they do not disappear when the block or function goes out of scope. A RETURN instruction does an implicit CLOSE so the latter won't appear very often in listings.

在顶层函数中，行[5]中的 CLOSE 使得虚拟机找到所有受影响的局部变量（它们必须是打开的 upvalue），把它们从栈中取出并放在安全的地方以便当块或函数超出作用域时它们不会消失。RETURN 指令做了隐式的 CLOSE 所以后者不会在清单中频繁出现。

Here is another example which illustrates a rather subtle point with CLOSE (thanks to Rici Lake for this nugget):

这里有另一个例子，它阐明了一个与 CLOSE 有关的相当微妙的地方（为此要感谢 Rici Lake）：

```
>do \
>>  local p \
>>  while true do \
>>    q = function() return p end \
>>    break \
>>  end \
>>end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 2 2
.local  "p"  ; 0
.const  "q"  ; 0

  ; function [0] definition (level 2)
  ; 1 upvalues, 0 params, 2 stacks
  .function  1 0 0 2
  .upvalue  "p"  ; 0
  [1] getupval   0   0       ; p
  [2] return     0   2
  [3] return     0   1
  ; end of function

[1] closure    1   0       ; 1 upvalues
[2] move       0   0
[3] setglobal  1   0       ; q
[4] jmp        1           ; to [6]
[5] jmp       -5           ; to [1]
[6] close      0
[7] return     0   1
; end of function
```

In the above example, a function is instantiated within a loop. In real-world code, a loop may instantiate a number of such functions. Each of these functions will have its own **p** upvalue. The subtle point is that the **break** (the JMP on line [4]) does not jump to the RETURN instruction in line [7]; instead it reaches the CLOSE instruction on line [6]. Whether or not execution exits a loop normally or through a **break**, the code within the loop may have caused the instantiation of one or more functions and their associated upvalues. Thus the enclosing **do end** block must execute its CLOSE instruction; if we always remember to

associate the CLOSE with the **do end** block, there will be no confusion.

上例中再循环内实例化一个函数。在现实的代码中，循环可以实例化很多这样的函数。这些函数中的每个都将具有自己的 **p** upvalue。（译注－多个闭包可共享同一个 upvalue，同一个 upvalue 必是原来的同一个局部变量，局部于循环内的变量在每次迭代时都是不同的，因此这里的 p upvalue 是同一个。）微妙之处是 **break**（行[4]的 JMP）不是跳到行[7]中的 RETURN 指令；而是到达行[6]的 CLOSE 指令。不论执行绪是正常退出循环还是通过 **break**，循环内的代码可能已经引起了一个或多个函数及其关联的 upvalue 的实例化。因此封闭的 **do end** 块必须执行器 CLOSE 指令；如果我们一直牢记把 CLOSE 与 **do end** 块关联起来将不会有混乱。（译注－看似内部带有闭包的程序块结束时，如果没有 RETURN 则会生成 CLOSE 作为循环结束后的第一条指令，而 break 是跳到该第一条指令处的，如此理解。）

CLOSE also appears when **for** loops are used in the same manner. When using loop indices or loop iterators as upvalues to instantiate functions, each instantiation will have its own unique upvalue. This is the expected behaviour in Lua 5.1 if loop indices or iterators are to be considered as locals to the loop body. Previously, Lua 5.0.2 considers loop indices or iterators to be local to a block enclosing the entire loop, and instantiation of multiple functions only results in a single upvalue shared between the functions. Please see the section on loop instructions for sample code that illustrates this behaviour.

当 **for** 循环用于同样的风格时 CLOSE 也会出现。当使用循环索引或循环迭代器作为 upvalue 来实例化函数时，每个实例将有其自己唯一的 upvalue。在 Lua5.1 中，如果循环索引或迭代器被视作局部于循环体的变量，那么这就是期望的行为。先前，Lua5.0.2 认为循环索引或迭代器是局部于包裹着整个循环的程序块的变量，并且多个函数的实例只导致在函数间共享单个 upvalue。请看循环指令示例代码部分，它们阐明了这种行为。